Complex event detection on an enterprise service bus

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

COMPLEX EVENT DETECTION ON AN ENTERPRISE SERVICE BUS

Master Thesis

Systems Group
September 16, 2008 – March 16, 2009

Silvio Kohler
ETH Zurich
kohlers@student.ethz.ch

Supervised by:
Dr. Peter Fischer
Kyumars Sheykh Esmaili

Abstract

Service-oriented architectures (SOA) have been adapted by businesses to improve their flexibility, recently with a focus on dynamic outsourcing of business processes. A cornerstone of a SOA is an Enterprise Service Bus (ESB), which is used to loosely connect services by the means of message exchange.
Businesses using a SOA still need to comply with applicable laws and regulations. To ensure this compliance, the MASTER [12] project supposes a control infrastructure to be implemented as another service in the SOA. Observing the message flow on an ESB provides ample opportunities to observe and compute indicators for compliance.

The control infrastructure mentioned above needs to catch raw messages on the ESB to detect complex events and answer queries posed by a higher level infrastructure. The goal of this master thesis is to close the gap between the interface of such a higher level infrastructure (defined by MASTER) and the interfaces provided by existing ESBs. This includes providing an overview of existing Enterprise Service Buses and the interfaces they offer. A model is developed which describes the interfaces and the mapping between them. A prototype for a selected ESB is implemented. The performance of the prototype is evaluated to identify challenges in processing large numbers of messages on an ESB.

Contents

1 Introduction **7**
- 1.1 The MASTER project . . . 7
  - 1.1.1 Architecture of MASTER . . . 7
  - 1.1.2 The role of ESBs in MASTER . . . 7
- 1.2 Scope and goal of this thesis . . . 8

2 ESB Overview **9**
- 2.1 General design . . . 9
- 2.2 Core capabilities of an ESB . . . 10
- 2.3 State of the art of ESB implementations . . . 11
  - 2.3.1 Existing ESB implementations . . . 11
  - 2.3.2 Classification . . . 12
  - 2.3.3 Message interception . . . 18
- 2.4 Choice of ESB for MASTER . . . 20
  - 2.4.1 Criteria . . . 20
  - 2.4.2 Evaluation . . . 23
  - 2.4.3 Experiences . . . 24

3 Modeling **27**
- 3.1 Goal of Modeling . . . 27
- 3.2 Signaling Interface . . . 27
- 3.3 ESB interfaces/interception points . . . 28
- 3.4 2-layer-model . . . 29
  - 3.4.1 Vocabulary . . . 29
  - 3.4.2 Description of 2-layer-model . . . 29
- 3.5 Using the model in MASTER . . . 32

4 Implementation **35**
- 4.1 Architecture of ServiceMix / JBI . . . 35
  - 4.1.1 The Java Business Integration specification . . . 35
  - 4.1.2 ServiceMix as a JBI implementation . . . 38
- 4.2 Interception in ServiceMix . . . 42
  - 4.2.1 Architecture . . . 42

4.2.2 Further implementation steps . . . 47
4.2.3 How to install and use the signaling prototype . . . 49

5 Performance **51**
5.1 Goal of performance experiments . . . 51
5.2 Test infrastructure . . . 52
5.2.1 Testbed . . . 52
5.2.2 Setting . . . 52
5.3 Experiments . . . 54
5.3.1 Experiment: Baseline . . . 55
5.3.2 Experiment: Signaling . . . 56
5.3.3 Experiment: Intermediate steps . . . 57
5.3.4 Experiment: Message persistence . . . 60
5.3.5 Experiment: Heap size and CPU usage . . . 62

6 Conclusion **65**

Bibliography **66**

A Example MEF event **69**

B ServiceMix configuration **71**
B.1 Installation checklist . . . 71
B.2 servicemix.xml configuration file . . . 72

C Testbed settings **75**
C.1 CXF configuration . . . 75
C.2 Operating system settings . . . 76

Chapter 1

Introduction

1.1 The MASTER project

Service-oriented architectures (SOA) have been adapted by businesses to improve their flexibility, recently with a focus on dynamic outsourcing of business processes. The MASTER [12] project aims to build control structures for SOA infrastructures to allow monitoring and enforcement of business processes in a SOA.

1.1.1 Architecture of MASTER

Business processes in a SOA infrastructure still need to comply with laws and regulations. The fact that the processes are distributed over several systems, which may reside on different machines with distributed ownership, introduces new challenges related to reliability, performance and security to ensure this compliance.
The MASTER project proposes an architecture based on dedicated components for observation, evaluation and reaction. Figure 1.1 illustrates these components.
The observation layer extracts raw events from the services in the SOA and generates complex events. The enforcement layer analyzes the complex events and provides reporting facilities and enforcement. The observation layer consists of two parts: The signaling component and the monitoring component. Signaling is responsible for dealing with different interfaces of the underlying services and providing the raw events, which the monitoring component aggregates to complex events.

1.1.2 The role of ESBs in MASTER

A cornerstone of a SOA is an Enterprise Service Bus (ESB). An ESB is used to loosely connect services by the means of message exchange. It provides facilities for services to connect and register themselves, and be discovered by other services. Another facility of an ESB is to ease communication between services by providing message transport infrastructures, and routing mechanisms for messages.

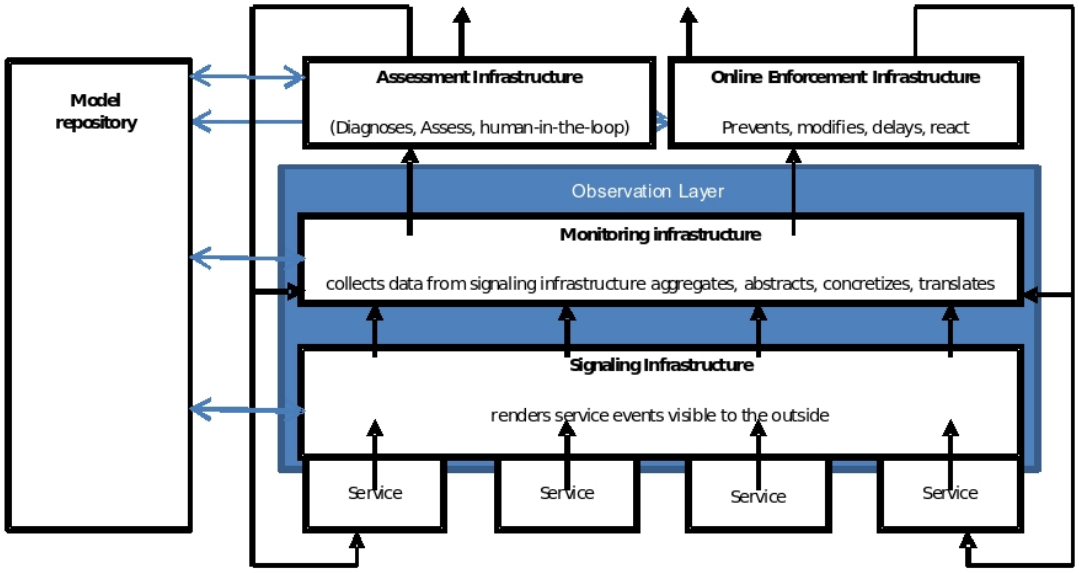

Figure 1.1: MASTER components, [19]

As an ESB is a key part in a SOA, it is in the scope of the MASTER project to look into it as a source of raw events for the signaling component.

1.2 Scope and goal of this thesis

Raw events, as introduced in section 1.1.1, that are exposed by an ESB, are mainly messages exchanged by services. Signaling on an ESB thus has to deal with a lot of messages flying by, filtering them on different criteria, and creating events that also contain metadata that belongs to the message.
This thesis' goal is to find a model that describes this step in a way that is suitable for all currently used ESB implementations. This includes providing an overview of existing ESBs and the interfaces they offer (chapter 2). From this, a model is derived which describes how abstract queries can be transformed to specific queries depending on a concrete implementation of an ESB (chapter 3). A prototype for a selected ESB is implemented (chapter 4). It uses the interfaces of the ESB to catch messages on the bus. The messages are processed in terms of filtering and examining their content to build events for the signaling interface. The evaluation of the prototype points out challenges in message interception concerning performance, and how these challenges can be dealt with (chapter 5).

Chapter 2

ESB Overview

An Enterprise Service Bus is used to connect existing and new software components to build up a Service Oriented Architecture. It needs to be able to connect to any IT resource, whatever its technology is or wherever it is deployed. The ESB needs to be flexible to easily combine and re-assemble components to meet changing requirements without disruption. It connects components in a loosely coupled way. This provides the ability to integrate existing systems into a SOA and deploy it step by step.

2.1 General design

The general architecture of an ESB with components connected to it is shown in figure 2.1. Components can take on the role of service producers (often called 'services') or service consumers. Services can be special components, such as orchestration engines, adapters to data resources or adapters to external systems which need message transformation or transport protocol conversion. The ESB mediates messages between the components, decides on where to route messages, and transforms message payload to fit the receiving components format. To achieve reliability, the ESB needs persistent memory, e.g. a database connected to it. For service consumers to be able to use the services provided by service providers, the ESB has to offer the facility to register them at the ESB and offer their interface to service consumers. This ability is referred to as service registering.

One approach to define a common architecture of an ESB is the Java Business Integration specification (JBI, section 4.1.1, [11]). Several implementations follow this specification.
The JBI specification describes a pluggable architecture for a container that hosts service producer and consumer components. Services connect to the container via binding components (BC) or can be hosted inside the container as part of a service engine (SE). Services are described using the Web Services Description Language (WSDL). Messages are always translated to a common message format and routed by the Nor-

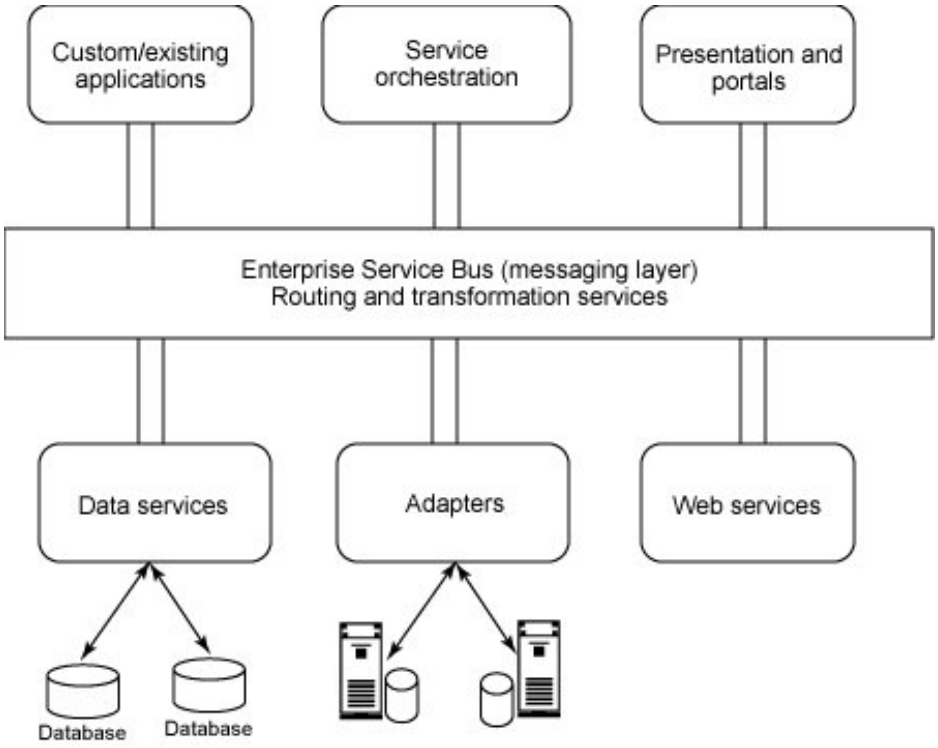

Figure 2.1: General ESB architecture [1]

malized Message Router (NMR). Four message exchange patterns (MEP) are specified: In-Only, Robust In-Only, In-Out, In Optional-Out.
A JBI-instance is bound to one Java virtual machine. To scale a JBI based system, several instances running on different machines have to be connected via binding components. This implicates several challenges, for instance of how to distribute a central service registry to let consumers know about services across JVM boundaries.

2.2 Core capabilities of an ESB

The term ESB does not denote a special product, but a type or group of products. They are defined by the capabilities they provide, and the tasks they fulfill. The following core capabilities of an ESB can be identified:

Routing The ESB can determine the destination component of a message based on different criteria, e.g. based on message content.

Transformation To ensure that other systems or external services can be attached to the ESB, it must provide the ability to transform these systems' message format to a format that is understood by the other components.

Transport An ESB has to be able to adapt the transport protocols connected components use and transmit messages according to the desired messaging pattern (e.g. asynchronous and reliable messaging).

Service registering/management Service consumers need to have the possibility to detect services attached to the ESB. It has to be possible to assemble composite applications from different services.

In addition, an ESB should have the following characteristics:

Scalability The topology of the ESB has to be designed in a way that it is possible to add as many components as wanted. No bottlenecks should evolve.

Reliability A message has to be transmitted reliably according to the selected messaging pattern. After a crash, the ESB can recover its previous state without data loss.

Existing ESB implementations follow very different models to achieve this functionality. They differ in the choice of possible topologies, how registering is implemented or where transformation is done.

2.3 State of the art of ESB implementations

Various products exist which fulfill the capabilities mentioned in section 2.2, and thus are used as ESBs. This section gives an overview of common open source products and proprietary ones.

2.3.1 Existing ESB implementations

Open Source ESB implementations

MuleESB [13] MuleESB, which is open source and supported by mulesource, is described as "a lightweight messaging framework and a highly distributable object broker designed around the ESB integration pattern". It has a very flexible architecture which allows to deploy any topology and distribute the whole ESB functionality over multiple servers. Routing can also be implemented without a central routing component, but by specifying destination endpoints directly at sending components. Mule components can be extended with interceptors. An interceptor is executed when a messages enters or leaves the component.

Sun OpenESB [16] OpenESB is a JBI based open source ESB implementation. An ESB distributed over multiple servers can be deployed by connecting several JBI instances in a star topology with a special JBI instance acting as a central point.

Apache ServiceMix [8] ServiceMix is another JBI based open source ESB implementation. It proposes JMS flow using ActiveMQ to combine multiple containers distributed over several servers to form a distributed ESB. Thus, like with OpenESB, any topology is possible.

Apache Synapse [9] Synapse is a simple, high performance ESB from Apache. It does not follow the JBI specification, but has a similar architecture. By connecting multiple instances using e.g. HTTP/SOAP, any topology can be deployed, although this is harder to implement compared to other ESB implementations.

Petals [14] Petals ESB is also JBI based. It provides different facilities such as a replicated registry which help to deploy clustered ESBs like e.g. with OpenESB. Binding components provided by Petals ESB can be extended with interceptors. An interceptor is executed in the sender binding component before a message is sent into the bus, or in the receiver component, when a message is delivered.

Proprietary ESB implementations

There are several commercial distributions of open source ESBs. These provide in general only additional support or performance warranties, and are thus not interesting from a technical point of view, as their open source correspondents are already covered. Other proprietary ESBs, in contrast, are interesting candidates.

BEA Aqualogic [10] Oracle's BEA Aqualogic Service Bus has a centralized topology. The bus resides in the center, and components (service providers and consumers) can connect to it. All ESB functionalities are performed at this central place. For scalability reasons, BEA Aqualogic Service Bus can be distributed over several physical servers, while the logical view does not change.

Progress Sonic ESB [15] The Progress Sonic ESB is very similar to BEA Aqualogic in its architecture. It differs only in the amount of message format transformation or transport components it provides.

In fact, most of the proprietary ESB implementations have a similar architecture. Interception on these ESBs can, without having access to source code, only happen at the interfaces provided to connect services. Internally, e.g. for the message queue, there are often proprietary protocols in use. Thus, for intercepting internal messages, for each ESB implementation, an own interception component must be produced.

2.3.2 Classification

For each task of an ESB, there are very different ways to fulfill it. Concerning the choice of these solutions, ESB implementations present a very heterogeneous scene. A classification of the ESB should thus start by analyzing these task dimensions separately.

The following tables show the capabilities of a sample of ESB implementations in terms of their environment (table 2.1), their functional capabilities (table 2.2) and their non-functional capabilities (table 2.3).
Maturity contains information about how emerged an ESB implementation is on the market and how many companies use it in production.
Important for the **scalability** is how the components of a SOA can be connected, i.e. which topologies are possible. Also, not every implementation provides load balancing on several components that implement the same functionality.
Reliability refers to the reliable transport of messages from one component to another.
Support is given in different manners. For some ESBs, there exists a commercial version, mostly with extended support.
Tools include functionality for the development of composite applications, and for managing and monitoring the ESB instance (e.g. statistics of messages passed or services connected).
For all open source implementations mentioned, Unit **Testing** is provided through JUnit. Some implementations also provide support for functional testing (test of whole service). Integration testing refers to the test of a composite application.
Routing is divided into two subcategories: The "how"-subcategory refers to the possible ways of deciding for the route (e.g. static, based on message content), the "where"-subcategory to if decisions are made on a central place or step by step on the route of the message (itinerary).
In most open source implementations, **transformation** can be done for any message format by implementing own transformation components. The most common are often already supported.
Messaging is done using lower level **transport** protocols. Some ESB implementations offer explicit support to implement Message Exchange Patterns (MEPs). Mule also allows message splitting to distribute tasks to different services.
Not every implementation supports the ability for **services** to **register** themselves on the ESB and thus making their functionality known to consumers automatically.
Message flow refers to the ability of intercepting messages at different stages of their transmission.

Most of the commercial ESB implementations have a centralized topology, where the ESB resides in the center and components are connected to it. To avoid bottlenecks, they can be spread over several physical servers. The grade of transparency of this distribution depends on the implementation.
Message transport between external components and the ESB can be controlled by its user. It should therefore be easy to intercept messages delivered over these channels. This in contrast to communication between internal components, such as BPEL-engines or transformation components. Closed source ESB implementations often do not provide explicit access to these messages (i.e. without knowing the often proprietary protocols used).

Table 2.1: ESB environment

		Mule ESB	**Sun Open ESB**	**Apache Synapse**	**Petals**	**Service Mix**	**BEA Aqua-logic (oracle)**	**Progress Sonic ESB**
Maturity	Stable version	Mule 2.0.2, 18/07/08	OpenESB v2, 03/06/08	Synapse 1.2, 09/06/08	Petals Platform v1.5.0, 22/09/08	Service Mix 3.2.2, 20/01/08	Service Bus 3.0	Sonic ESB 7.6, 06/05/08
	Market	1000	0, com-mercial release 11/08	several 100 (for WSO2)	?	?	-	-
	Refer-ences	Walmart, Netcetera, Deutsche Bank	-	care organiza-tion, auto maker (for WSO2)	-	-	T-Mobile, B-Source	Limbic systems, pacific blue cross
Support	Manual/ Tutorial	Y	Y	Y	Y	Y	Y	Y
	Mailing lists/Fo-rum	Y	Y	Y	Y	Y	Y	Y
	API	Y	Y	Y	Y	Y	N	N
	Comm. support	Y	Y	Y	N	N	Y	Y
Tools	IDE	Y (com-mercial version)	Y	N	Y	Y	N	Y
	Moni-toring/ Man-agement	N	Y	N	Y	Y	Y	Y
Testing	Unit testing	Y	Y	Y	Y	Y	N	N
	Functional testing	Y	N	N	N	N	N	N
	Integration testing	Y (help of moni-toring tool)	N	N	N	N	Y	Y

Table 2.2: Functional capabilities

		Mule ESB	**Sun Open ESB**	**Apache Synapse**	**Petals**	**Service Mix**	**BEA Aqua-logic (oracle)**	**Progress Sonic ESB**
Routing	Static	Y	Y	Y	Y	Y	Y	Y
	Content based	Y	Y	Y	Y	Y	Y	Y
	Workflow (i.e. BPEL, central)	supported	Y	Y	Y	Y	Y	Y
	Itinerary	Y	N	N	N	N	N	N
Transformation	possible	any	any	any	any	any	only supported transformations	only supported transformations
	supported	from and to: XML, byte arrays, hex strings, Stream, Object,...	XML	XML	XML	XML	XML, MFL	XML to many (non-) XML
Messaging/Transport	Possible protocols	any	any (outside JBI)	any	any	any (outside JBI)	only supported protocols	only supported protocols
	Supported protocols	JMS, FTP, TCP, UDP, HTTP, IMAP, JDBC	File, FTP, HTTP, JMS, Mail, SIP, TCP, JDBC	HTTP, HTTPS, JMS, SMTP, JDBC	File, FTP, HTTP, JMS, Mail, JDBC	File, FTP, HTTP, JMS, Mail, JDBC	File, FTP, HTTP, JDBC, HTTPS, JMS, Mail, Inter-ESB (proprietary)	File, FTP, HTTP, HTTPS, JMS, Mail, JDBC

		Mule ESB	**Sun Open ESB**	**Apache Synapse**	**Petals**	**Service Mix**	**BEA Aqua-logic (oracle)**	**Progress Sonic ESB**
Messaging/Transport	MEPs	Y (needs appropri-ate transport protocol)	Y	Y	Y	Y	N	N
	Message splitting and ag-gregation	both imple-mented	N	N	N	N	N	both imple-mented
Service registering	auto matically	N	Y (inside JBI)	N	Y (dis-tributed)	Y	Y	Y
	manually	Y	Y	Y	Y	Y	Y	Y
Message flow	read message at different stages	Y, re-strictions depend on transport used	Y	Y	Y	Y	N	N
	transform message at different stages	Y, no re-strictions	Y	Y	Y	Y	N	N

Table 2.3: Non-functional capabilities

		Mule ESB	**Sun Open ESB**	**Apache Synapse**	**Petals**	**Service Mix**	**BEA Aqua-logic (oracle)**	**Progress Sonic ESB**
Scalability	Topology	any possible	central in JVM-instance, any outside JVM-instance	central	central in JVM-instance, any outside JVM-instance (special support for distri-bution)	central in JVM-instance, any outside JVM-instance	central	central
	Concurrent connec-tions	limited (100), blocks on receipt	unlimited	unlimited	unlimited	unlimited	unlimited	unlimited
	Load balancing	Y	N	Y	Y	Y	Y	Y
Reliability	Transport	Y, depends on transport protocol	Y	Y	Y	Y	Y	Y

Management and monitoring tools can be used to get information about services registered. These tools will be a good starting point to get runtime information. A central service registry may also help for this purpose.
ESB implementations differ, also dependent on their topologies, in where and how they make routing decisions or transform messages. An abstract view of these mechanisms must be found that matches all ESB implementations to find a common way of intercepting.

2.3.3 Message interception

Table 2.4 shows for several ESB implementations, where particular information, which needs to be provided on the signaling interface, can be obtained.

Connected services Information about connected services includes their names, their functionality and their interfaces.

Transformations Includes information about which transformations are possible at which states in message transmission. This may be important to determine where messages have to be intercepted.

Message contents Information that is kept in messages, such as the destination address in the message header. Their content depends generally on the state of their transmission, i.e. if they have been transformed or if they have passed a router component, which could have changed routing information.

Routing This information is primarily useful to trace a message on its path to its destination endpoint, or to forecast its destination, respectively. Routing decisions are not in all cases taken at one central point, but can also be taken at e.g. endpoints.

The heterogeneity of the ESB landscape makes it harder to find a common way of intercepting. One concept that every implementation uses, although in it's own way, is the concept of virtual endpoints.
A virtual endpoint shields the user of a service from the actual endpoint. It uses no hard coded URL, but hides the address of the actual service to enable loose coupling of components. Another application of this concept is to scale a service up to multiple hardware devices.
All messages sent from and to services pass at least one endpoint. Observing endpoints for interception should thus allow to catch all messages sent in a SOA connected with an ESB.
To get information about the status of the ESB itself, e.g. about services that are connected, the component where they are registered, i.e. the registry, must also be observed. Not every ESB implementation contains a centralized registry. In some cases, it will be necessary to gather this information from several, distributed places.
Some ESB implementations use queues to implement asynchronous messaging, mainly

Table 2.4: Interception points

	connected services	**transformations**	**message contents**	**routing**
MuleESB	xml configuration file	xml configuration file. A transformer can be any component (java-class, xslt file) -> difficult to extract information	3 states: before leaving outbound router, on bus (transport), after passing inbound router (includes filtering). Content/type depends on transformers and filters attached to router (no common message format).	xml configuration file. Also as properties of in- and outbound router. BPEL (BPEL-engine available)
JBI based ESBs	NMR-interfaces (javax.jbi)	done by dedicated SE which uses XSLT (only need to transform XML-payload –> WSDL). Special transformation SEs can be deployed.	Normalized Message consists of XML-payload (which can be transformed by transformer-SE), header (transaction contexts, security information) and attachment (any binary data)	BPEL-Engine (which is an SE) or dedicated routing engine (which can be any SE)
Apache Synapse	general: XML based configuration language	XSLT	SOAP message	XML file (configuration language)

for internal messaging with JMS. Although, messages to external components are still sent using i.e. HTTP, so, in most cases synchronous and without queuing. For this reason, intercepting queues is not a common possibility to get all messages delivered. Observing routing engines, provided they are central, are a further way to get information about the ESB instance. As routing functionality is often distributed to several components, it will be necessary to decide for a particular implementation if it is necessary at all to observe it, and how to do it.

2.4 Choice of ESB for MASTER

2.4.1 Criteria

The ESB implementation which is chosen to build the prototype of an interception component should fulfill the following criteria best.

Representative The implementation should use the most common concepts that are interesting for interception to show their usability for this purpose. Among them is the concept of virtual endpoints, centralized registry, and a centralized routing engine.

Maturity The ESB implementation must be stable and "tested" in the sense that it is used successfully in productive systems. It must implement all commonly used transport protocols and message transformations.

Support, documentation A good documentation which explains the architecture of the ESB and helps configuring and managing it together with support help to use all features of the ESB and make interception fast and complete.

Reliability Message transport in the ESB must be reliable to ensure that all messages can be caught. In addition, it is important to know about how reliability is achieved (i.e. using persistent queues), and which grade of reliability is possible (can a system recover after crash or only ensure that no double messages are delivered?).

Features Additional features such as management and monitoring tools provide easy access to information about services registered and messages sent over the bus.

Topology Routing, transport and service registering, which correlate closely with the topology of the ESB implementation, must be considered to decide on the interception model. First in the abstract concept and later in the concrete implementation.

Performance The chosen ESB implementation should have adequate performance numbers. This issue is discussed in the following chapter.

Table 2.5 contains information about the criteria mentioned above that are not yet covered in this document: Support, documentation and reliability.
Another important criterion is performance. As the settings of ESB performance tests differ, the results can not be compared across all ESB implementations. The listings below show numbers from tests that include most of the ESB implementations discussed. From these results, performance numbers for an adequate implementation can be derived.
Table 2.6 shows results (transactions per second) from a test performed by WSO2/ Apache [17]. Three scenarios were deployed:

1. The ESB simply serves as a proxy server for a single service which is used by a number of clients.

2. The ESB routes messages based on their XML-payload.

3. The whole message payload is transformed by the ESB.

For this test, small messages were used (1 KB).
Table 2.7 shows response times for the last two settings for a 1 KB message.
Another test includes commercial ESB implementations [2]. It uses large size messages (MB range). The size of the messages leads to performance degradation due to message serialization.
The test used the same scenario as the WSO2 test (ESB as proxy) for 1 to 50 clients. For the big messages used, no dependency on the number of clients has been observed.
Table 2.8 shows the response time for 1 MB messages in seconds.
Table 2.9 shows the response time in seconds depending on the message size for the same setting.
The test results can not directly be compared or merged for already mentioned reasons. Although, reasonable performance numbers can be derived that an ESB implementation must fulfill.

- 100's up to 1000 1 KB messages should be processed in a proxy setting, also for big number of concurrent clients (requests not blocking).

- Same performance when simple XML-content based routing is used.

- 100's of 1 KB messages should be processed when simple XSLT transformation is applied.

- Response time for 1 KB messages should be in any case smaller than 1 second. It should not be greater than 10 ms without transformation and few clients.

- For 1 MB messages, response time should be in the range of few seconds (approximately 10 seconds).

These numbers must be verified for an ESB implementation that is chosen to use, using a simple scenario with an echo-service.

Table 2.5: Decision criteria

	Support community, documentation	**Reliability, Scalability**
MuleESB	Full Javadoc available. Active community with forum, examples, mailing lists with a lot of information on the many components.	Scalability: Showed poor scalability in terms of concurrent connections in a test made by WSO2 (Synapse), but scalable topology possible. Reliability: No dedicated mechanism, use reliable transport (persistent, e.g. JMS) or idempotent operations.
Apache ServiceMix	Full Javadoc available. Lack of information about architecture on project page. Found on other pages. Active community with forum, examples, mailing lists.	Scalability: "Mesh of bus"-topology provides scalability for distributed systems. Reliability: NMR for internal transport, use e.g. JMS for external transport. No persistent orchestration engine.
Apache Synapse	Full Javadocs available. Lack of information about architecture. Active community with forum, examples, mailing lists.	Scalability: Best results in performance tests made by WSO2 (Synapse). "Mesh of bus"-topology provides scalability for distributed systems. Reliability: Internal transport with queues.
OpenESB	Full Javadocs available. Active community with forum, examples, mailing lists.	Scalability: "Mesh of bus"-topology provides scalability for distributed systems. Reliability: NMR for internal transport, use e.g. JMS for external transport. Database persistence for BPEL SE.
Petals	Full Javadocs available. Active community with forum, examples, mailing lists.	Scalability: "Mesh of bus"-topology provides scalability for distributed systems. Reliability: NMR for internal transport, use e.g. JMS for external transport.

Table 2.6: WSO2 results. TPS for 1 KB messages

Number of clients	20	40	80	160
ESB as proxy				
WSO2	1400	2400	3400	4200
Mule	580	590	610	crashed
ServiceMix	1140	1100	1220	1380
Proprietary	1400	2800	4700	5000
Content based routing				
WSO2	1210	1870	2040	2100
Mule	570	590	560	crashed
ServiceMix	560	690	780	690
Proprietary	1180	1570	1510	1530
XSLT Transformation				
WSO2	540	560	530	470
Mule	600	570	590	crashed
ServiceMix	600	650	600	580
Proprietary	870	880	920	890

Table 2.7: WSO2 results. Response time for 1 KB messages

Number of clients	20	40	80	160
Content based routing				
WSO2	8	14	26	58
Proprietary	10	18	36	76
XSLT Transformation				
WSO2	20	32	75	175
Proprietary	37	75	160	310

2.4.2 Evaluation

Representative As JBI for the far is the most common specification used in open source ESB implementations, it is suggested to limit the choice to JBI based ones. This covers the largest group of implementations.

Maturity All ESB implementations in this survey are used productively and support the common transports (FTP, HTTP, JMS, JDBC) and message transformations (XML).

Mule and Petals have a special feature which eases the interception at endpoints: Interceptors are filters that are executed before and after the event/message is processed (Petals only at binding components).

ServiceMix has a listener interface which can be used for the same purpose.

Table 2.8: Response time for 1 MB messages

Number of clients	1-50, no dependency
IBM Websphere	6.5
BEA Aqualogic	5.0
Oracle	4.5

Table 2.9: Response time depending on message size

Message size (KB)	200	800	1400
IBM Websphere	5	14	25
BEA Aqualogic	3	8	14
Oracle	3	8	14

Support, documentation Support and documentation seems to be mature for all presented open source ESB implementations. Only ServiceMix does not have satisfiable information about the architecture on their website.

Reliability Each of the implementations offers reliable message transports (e.g. JMS with queues) that can be chosen for reliable transport. Queues are in most cases in memory, but can be made persistent by just setting a flag.
This does not yet ensure reliability in the sense of that the whole state of the ESB, i.e. the composite applications, can be recovered after a crash of the server hosting the ESB.

Features Open ESB has a monitoring and editing tool integrated in NetBeans. Petals offers the same functionality as an eclipse plug-in. ServiceMix uses a JMX based monitoring tool from J2SE (JConsole). This is not special SOA monitor.

Topology The "Mesh of Bus"-Topology most JBI based ESB implementations use is very common. The star-topology with a central node handling all ESB-functionality is often applied in commercial implementations. As the "Mesh of Bus"-Topology combines the star- and the mesh-topology, this choice covers aspects of both architectures.

Performance As mentioned in the preceding chapter, performance numbers have to be checked for a chosen ESB implementation.

Among JBI based Open Source ESB implementations, ServiceMix, OpenESB and Petals seem to fulfill the above requirements best. For these reasons, these three ESBs are installed and tested.

2.4.3 Experiences

The three ESBs which were tested have the following characteristics:

OpenESB OpenESB has a handy, though quite slow graphical user interface. Thus, a business process developer does not need to write code, not even XML-configuration files. It is aimed at being a well working platform to develop business processes, without providing much facilities for developing special applications.

Petals Petals is similar to Open ESB, but uses a web interface instead of the more sophisticated OpenESB user interface.

ServiceMix ServiceMix focuses more on development. It has different components that provide similar functionality to test different approaches. Internal interfaces provide much functionality, such as a listener interface with different types of listeners. Though, for deploying services, no graphical user interface is available. Also, the components have a lot of bugs.

Despite the drawbacks concerning usability and maturity of components, ServiceMix provides a good basis for developing the prototype due to the listener interfaces and the different types of components used.

Chapter 3

Modeling

3.1 Goal of Modeling

The goal of the modeling presented in this chapter is to describe the signaling step of MASTER in a formal manner, using ontologies, as required by MASTER. The abstract interface of the signaling step is formalized as well as the core capabilities of an ESB and the lower, implementation specific architecture with ESB components. Interception points for obtaining raw events from the ESB are described in the model, combined with a mapping to the signaling interface to show how to get the required information from the actual ESB implementation.
To obtain this model, the components and relations between them are described in the lower level, implementation specific models, and commonalities of different implementations in a higher level model. These models are called the 'How'-Models, as they describe how the ESB functionality is achieved. The interception points identified in section 3.3 are analyzed and depicted in both the lower level How-Models and the higher level How-Model, as well as the mapping between the to levels. To illustrate the core capabilities of ESBs as introduced in section 2.2, another model, the 'What'-Model, is developed and mapped to the How-Model. This is called the 2-layer-model, which is explained in detail in section 3.4.

3.2 Signaling Interface

Monitoring and enforcement in MASTER needs information about the business processes in the SOA in the form of events. In the case of an ESB, this information consists of the less dynamic 'status' information and the more dynamic 'runtime' information. Status information is information about registered services and their endpoints, and about routing and transformation information. This may change, also at runtime, but not very often. Runtime information consists of the messages services exchange, and metadata belonging to the messages, such as the time the message flew by, and the services the messages were sent from or to.

3.3 ESB interfaces/interception points

Concerning the design of existing ESB implementations, to monitor an ESB, several components of it must be observed. In the following, possible interception points for components are listed, with notes on which actions can be observed and which information they can provide.

- Virtual Endpoint

  Description Builds an interface for real endpoints/services. Services are invoked using virtual endpoints. The real endpoint is hidden. Is used to bind service names flexibly to their address. Can have load balancing functionality, but not routing (route message to one of equal services without changing message content).

  Actions Receive message, reply of service at endpoint, forward to other endpoint (possibly with changed payload), endpoint fault reply

  Messages Message contents (headers+payload) of message received/sent in actions

  Metadata Id of endpoint, date/time, correlation id for actions on 'same' received message, next receiver (endpoint and/or intermediate queues)

  Events For each action, generate an event with the observed message and metadata.

- Transformation component (MessageTransformation)

  Description Change of the form or content of the message payload

  Actions Similar to endpoint, without service reply or unchanged message

  Messages Similar to endpoint

  Metadata Id of transformation component, date/time, correlation id for actions on 'same' received message, transformation schemes, original/result formats, dropped or projected elements (if any)

  Events Similar to endpoint

- Registry

  Description Components where data about registered services (or partition of registered services) can be discovered by at least some of the service consumers. Data can be distributed over multiple components or kept central.

  Actions Service registration, deregistration (explicit or by failure)

  Messages Service description

  Events Service change + Service description

3.4 2-layer-model

As mentioned in section 2.3.2, modeling the existing ESB implementations in a hierarchy does not make sense because of the heterogeneity of the approaches used in the design of the ESBs. This hierarchy would consist of one root model - a very abstract model that fits all implementations - and several direct subnodes, one for each of the implementations. Instead, a model with two kinds of orthogonal layers is proposed. One layer, the What-Layer, describes the concepts of a general ESB, i.e. the common capabilities of ESBs as described in section 2.2. On the other layer, a model describes how these capabilities are achieved. This model, the general How-Model, can then be mapped to specific How-Models for each ESB implementation. There has to be developed a specific How-Model for each of the ESB implementations because of the differences in their architecture. If a mapping between the general How-Model and a specific one exists, the specific model can be hidden, and the ESB instance can be abstracted to the general How-Model. This approach is presented in detail in chapter 3.4.2.

3.4.1 Vocabulary

Even if the different ESB projects in some parts use the same structures and design, they use different terms to identify them. This might be confusing. For this reason, the most common terms, as they are used in the models, are defined in table 3.1.

3.4.2 Description of 2-layer-model

The upper layer of the proposed 2-layer-model is called the 'What'-Layer or 'What'-Model. It describes the core functionalities an ESB is supposed to provide. The lower layer, called 'How'-Layer, describes the basic structure of ESB implementations. As these implementations differ, a refinement of the How-Layer is needed for each product.
The groups of components of the ESBs can be arranged in taxonomies. This can be modeled with ontologies. Additionally, the dependencies between these groups can be described using relations in the ontology.

What-Layer

The core functionalities this layer describes are message routing, message transport, message transformation and service registering. Figure 3.1 depicts the What-Layer of the model. In addition, the basic entities in an ESB are shown and their relationship to the core functionalities. ConnectionPoint refers to an endpoint that represents a service in the ESB. An external service uses connection points to connect to the ESB. As messages represent the main interception information, the relationships to the core functionalities are shown as well.

Table 3.1: Vocabulary used in the How-Model

Concept	Definition
Abstract ESB	The abstract view of the ESB as a concept, defined by its functionality. These are message transport, routing, message transformation and service registering/managing.
ESB node	Smallest set of components that together provide full ESB functionality. Corresponds to a JBI instance in case of a JBI implementation, which is then bound to one JVM.
Centralized ESB	ESB with a star topology, where all messages pass a central node, which implements all ESB functionality. This logical ESB can be distributed over multiple servers, but this must be transparent.
Mesh of services	ESB that can be deployed in any topology. Components/Services can connect directly to other components (peer to peer).
Virtual endpoint	Builds an interface for real endpoints/services. Services are invoked using virtual endpoints. The real endpoint is hidden. Is used to bind service names flexibly to their address. Can have load balancing functionality, but not routing (route message to one of equal services without changing message content).
Transport	Direct message delivery to a known destination. Without routing functionality. Defined primarily with the transport protocol it uses.
Queue	Part of (asynchronous) transport. Can be used for routing facilities, but has itself no routing functionality.
Internal service	Service provider or consumer which is local to an ESB node from its own point of view. Differences to external services are in type of transport they use.
External service	Service provider or consumer which is not an internal service.
Protocol translator	Internal service component which provides the ability to change from one transport protocol to another to let external services connect to an ESB. E.g. a binding component as defined in the JBI specification.
Registry	Components where data about registered services (or partition of registered services) can be discovered by at least some of the service consumers. Data can be distributed over multiple components or kept central.
Transformation	Change of the form or content of the message payload.
Normalized Message	Message format specified in the JBI specification and used in the Normalized Message Router.

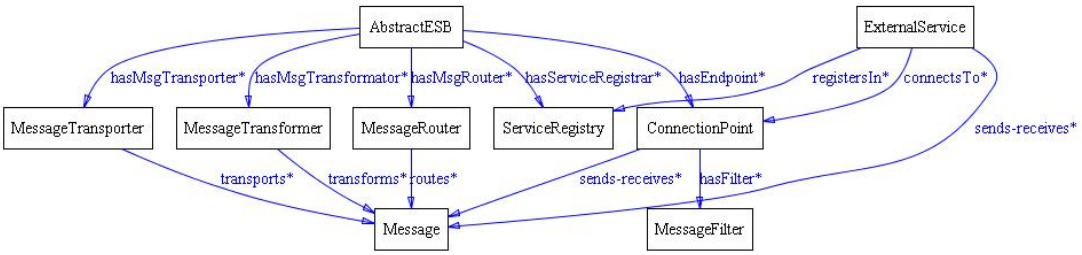

Figure 3.1: What-Layer of the 2-layer-model

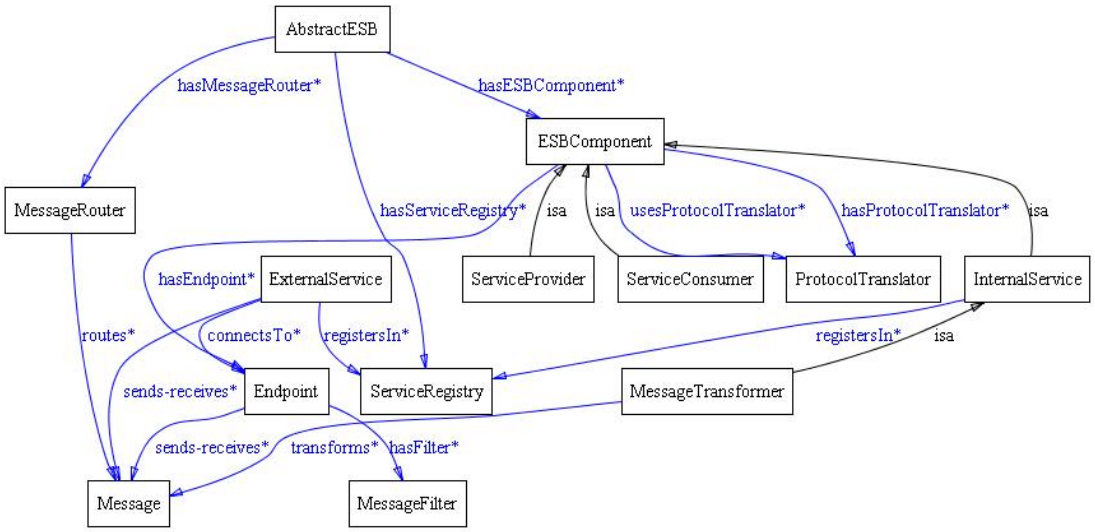

Figure 3.2: How-Layer of the 2-layer-model

How-Layer

The How-Layer, as shown in figure 3.2, provides a view on concepts and structures generally used in current ESB implementations. It includes the shared entities among them without sacrificing the abstraction.
An AbstractESB has components (services), which are categorized in two orthogonal dimensions. One dimensions denotes if the service is an internal or an external one, the other if the service is a provider or a consumer. ServiceRegistry, MessageRouter and ProtocolTranslator may, depending on the implementation, be services too, but not in the general case.

Specific How-Models As already mentioned, the How-Layer shows an abstraction of the actual implementations. Specific How-Models are refinements of the model of the How-Layer. As the general How-Model represents the signaling interface, while the specific How-Model represents the actual ESB interface, the matching between

these two models can serve as a guideline for the implementation of the signaling component.
Figure 3.3 shows the mapping from the general How-Model to the specific How-Model for Apache ServiceMix.
As ServiceMix is a JBI implementation, it has a special case of internal services: binding services. They are used to let external services connect to the ESB. Message transformation and message routing is implemented as internal services.
The ESB can be distributed over several physical machines. One ServiceMix instance is called an ESB node. The nodes can be connected in any topology. Together, they form a mesh of ESB nodes.
The ellipses in figure 3.3 depict which compound from the general model are mapped to which parts of the specific model. For example, the ESB component with its subtypes from the general How-Model is mapped to the corresponding taxonomy in the specific How-Model.

Mapping between What- and How-Layer

The mapping between the What- and the How-Layer shows how the functionalities in the What-Layer are implemented. In figure 3.4, an arrow pointing from a functionality in the What-Layer to an entity in the How-Layer indicates that the entity is involved in fulfilling the corresponding functionality.
Message transport is done by a dedicated component, or by each component by using their endpoints. For message transformation, a special internal service is responsible. Routing is carried out by the message router with the help of endpoints and the service registry. Service registry functionality is accomplished using the endpoints of the services and a dedicated registry.

3.5 Using the model in MASTER

The model presented can be used to ease the signaling and monitoring procedures within the MASTER project. In figure 3.5, the interception points that are used to obtain information for the signaling interface from the ESB are highlighted in the general How-Model. The following information can be obtained from the corresponding points:

1. Registered services and their interfaces.
2. Messages (metadata and content) as they enter/leave the component (through the endpoint).
3. How messages are routed (e.g. based on which criteria).
4. Message content/structure before/after transformation (as more dynamic information), or the transformation rules (as less dynamic information).

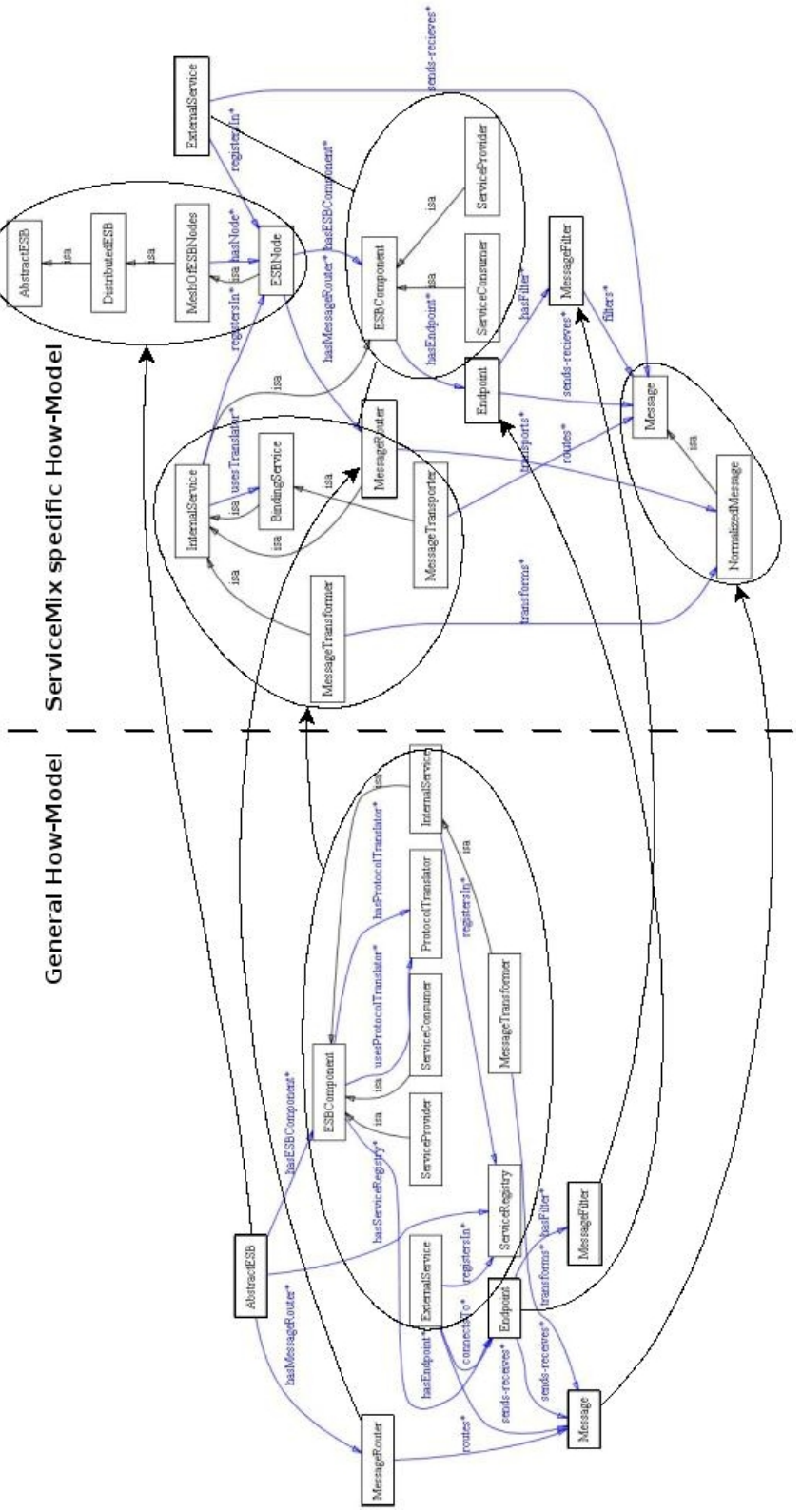

Figure 3.3: Mapping from general How-Model to ServiceMix specific How-Model

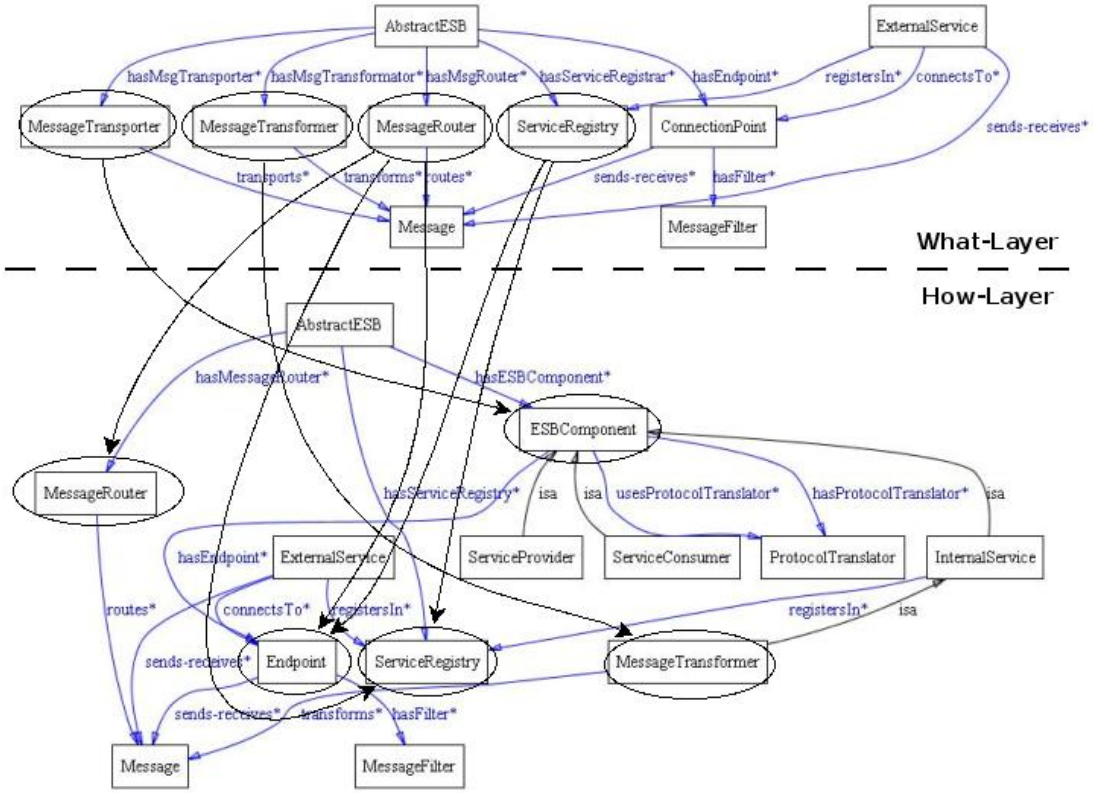

Figure 3.4: Mapping from What-Layer to How-Layer

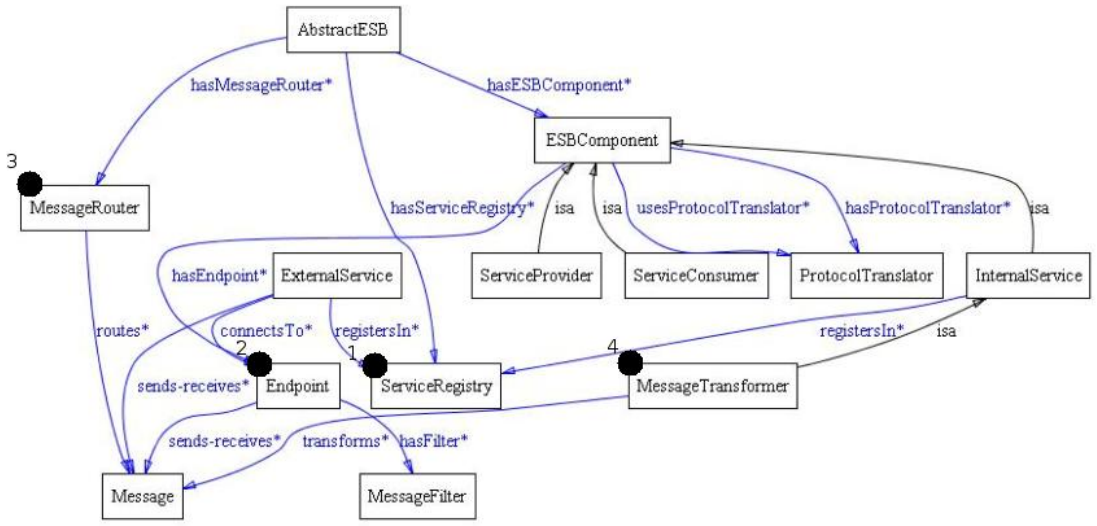

Figure 3.5: Possible interception points, depicted in the general How-Model

Chapter 4

Implementation

4.1 Architecture of ServiceMix / JBI

Apache ServiceMix, which is chosen to build the prototype (chapter 2), implements the Java Business Integration specification (JBI). Thus, pure JBI, as it is also used in other JBI based ESBs, is explained first (section 4.1.1). ServiceMix extends the specification with a few interfaces and implementations which are very useful for interception. These additions are explained in section 4.1.2.

4.1.1 The Java Business Integration specification

The basic structure of JBI is illustrated in figure 4.1.

Service Engines and Binding Components, which correspond to internal services in the How-Model presented in chapter 3, are connected to the Normalized Message Router, which represents the central bus, via Delivery Channels. The Normalized Message Router routes Normalized Messages, whose format is part of the JBI specification.
The message exchange JBI components (Service Engines or Binding Components) use to interact is described using Web Services Description Language (WSDL).
Services in JBI are wrapped into service units. One or more services can be in one service unit, but they all need to use the same component as connector to the container. One or more service units can be packaged in a service assembly as one jar-file, which can then be deployed to the container.

JBI messaging

The basic elements of the JBI messaging infrastructure are NormalizedMessage, MessageExchange and DeliveryChannel.

NormalizedMessage This interface defines the container which holds messages sent through the bus. A NormalizedMessage can be a service invocation message, a

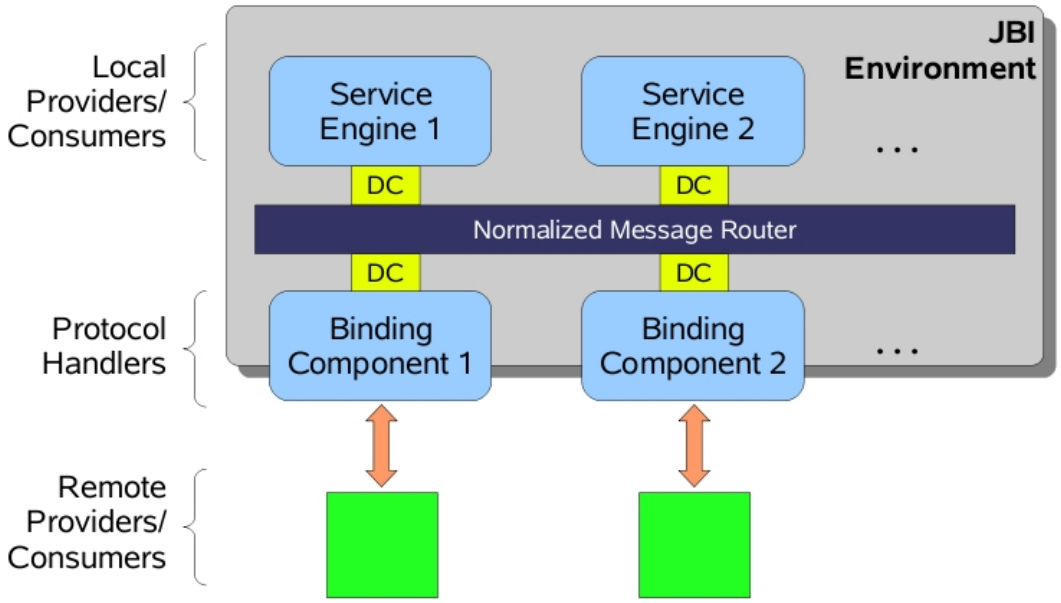

Figure 4.1: Basic structure of JBI [18]

response message as well as a fault message, which are all XML messages. It also includes message properties and a binary attachment. The properties are, depending on the configuration, also used for transporting e.g. SOAP-Headers in a common format.

MessageExchange This is a container which holds the message and the state of the service invocation (e.g. ACTIVE, DONE, ERROR). There are subtypes for each supported message exchange pattern (see section 4.1.1, Message Exchange Patterns). The possible values for the state depend on the message exchange pattern used.

DeliveryChannel Each component has a delivery channel, which is used to send and receive message exchanges. The delivery channel can be used to catch the message exchanges for interception, either by calling a blocking 'accept' method on its implementation or by letting it inform registered listeners.

The logic to route message exchanges is provided by the JBI implementation. This is referred to as the Normalized Message Router (NMR).

Message Exchange Patterns

Each message exchange in JBI has a particular message exchange pattern (MEP) bound to it. The MEP defines the ordering of message exchanges that occur when an operation is invoked. The MEP used by the ServiceMix-CXF binding component

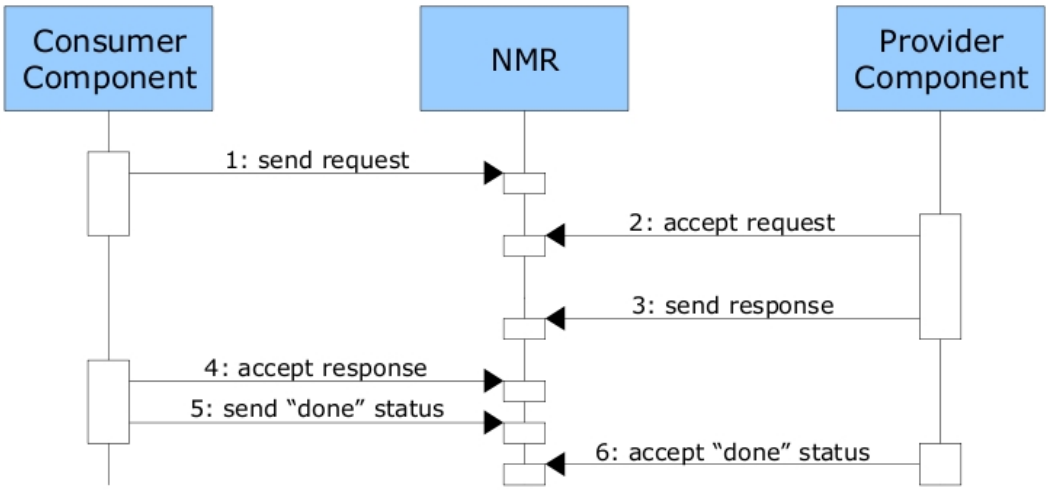

Figure 4.2: In-Out MEP, normal case [18]

[3] for the testbed explained in chapter 5 is In-Out. Figure 4.2 illustrates the normal response case for this pattern exemplarily.

The consumer creates a MessageExchange instance, sets the 'in' message to the request, and sends the instance to the NMR, which puts it in the queue appropriate to the provider component. The provider processes the request, sets the 'out' message to the response, and sends the MessageExchange instance to the NMR. The consumer accepts the instance and sets its status from 'active' to 'done'. The 'done' status indicates the end of the MEP. Thus, the In-Out MEP consists of three exchanges.
There are three more MEPs available:

In-Only The In-Only MEP is a one-way pattern. The consumer sends a MessageExchange instance with the 'in' message set. The provider accepts the exchange, sets its status to done and sends it back. The 'out' message is not set. Two exchanges are sent when this pattern is used.

Robust-In-Only The Robust-In-Only pattern extends the In-Only pattern by the option of sending a fault message back. If no fault occured, it is the same as the In-Only MEP. If the provider wants to send a fault, it is the same as the In-Out MEP, but with the response message set to a fault message. Two to three exchanges are sent.

In-Optional-Out In this pattern, the provider can respond to a request either with a normal message (then the pattern corresponds to In-Out), with a fault message (In-Out in case of fault), or with the status of the exchange set to 'done' (In-Only). In this pattern, two to four message exchanges take place.

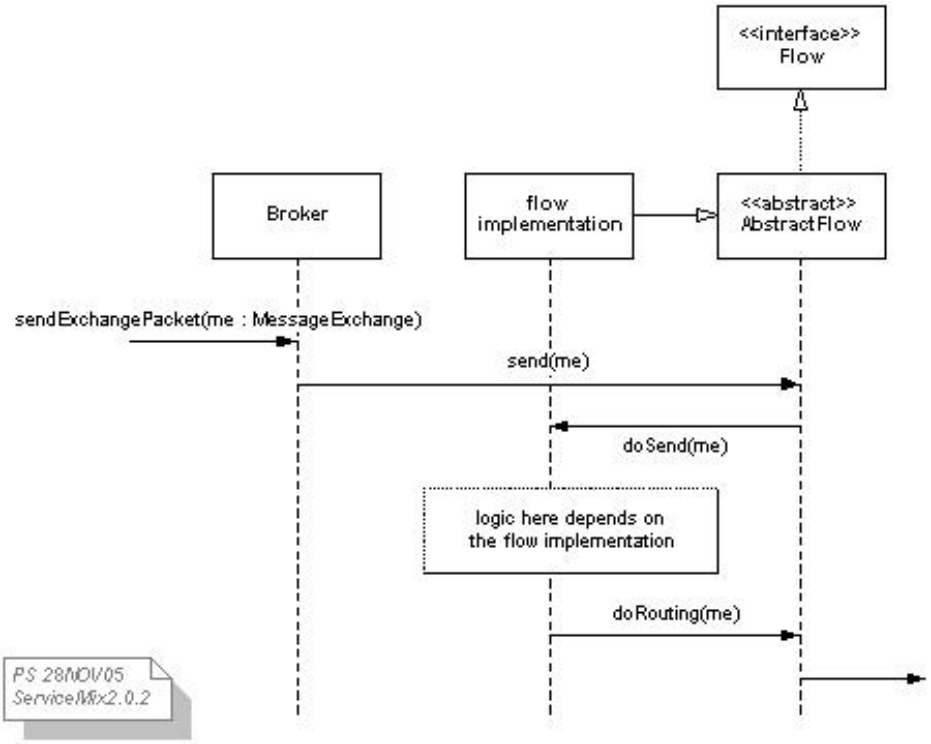

Figure 4.3: General procedure for flows in ServiceMix [4]

4.1.2 ServiceMix as a JBI implementation

Apache ServiceMix implements JBI, and extends it with mechanisms that are helpful for message interception.

For the implementation of the NMR, ServiceMix has four different flows. Flows are responsable for transporting the MessageExchange instance from the sending to the receiving service. When a service starts a message exchange, its delivery channel instance first calls each ExchangeListener's exchangeSent() method registered at the container. Then, it passes the exchange to the container, which uses a broker to choose a flow. Figure 4.3 illustrates the further procedure. The broker calls the send method of the AbstractFlow, the doSend() method of the actual flow implementation then determines how the message is transmitted. The doRouting() method of AbstractFlow chooses the appropriate endpoint and calls the accept() method of the corresponding delivery channel.

The flow that should be used to transmit a message can be chosen by setting the FLOW_PROPERTY_NAME property of the message exchange. If not set, the flow is chosen by the broker depending on the components default flow. The default flow for the CXF-component for example is SEDA flow. If this flow is not available, JMS flow is chosen.

Below, the flows available in ServiceMix are described. It is also explained how the message exchange is processed.

Straight Through flow The Straight Through flow is the simplest mechanism. It does not use staging or buffering. The doSend() method of the flow directly calls doRouting().

SEDA flow The SEDA (Staged Event Driven Architecture) flow uses in-memory queues for staging. It is illustrated in figure 4.4. The flow enqueues the exchange in a SedaQueue instance and returns from the doSend() method. Thus, the flow is available for new exchanges. The exchange is dequeued by another thread, which then calls the doRouting() method of AbstractFlow. This flow is fast and non-blocking.

JMS flow The JMS flow uses the Java Message Service (JMS) implementation Apache ActiveMQ's persistent queues. It can be seen as the persistent, thus slower variant of the SEDA flow. It is also used for communication among a network of remote containers. Figure 4.5 illustrates the JMS flow.
For this flow, there is one JMS queue per container and one JMS queue per component. The JMSFlow instance is also a listener for the container-queue and the component-queues. Requests are sent directly to the receiving components queue. The remote JMSFlow instance's (which may be the same) onMessage() method is triggered, which calls the doRouting() method of AbstractFlow. Responses are sent to the container-queue, where again the JMSFlow instance is listening for messages. The normalized message is wrapped in a JMS ObjectMessage for delivery.

JCA flow The JCA flow uses JCA 1.5 inbound resource adapters for message routing among a network of remote containers. The resource adapters encapsulate JMS functionality.

Detailed class and sequence diagrams can be found on the ServiceMix webpage [4].

ServiceMix, in addition to the implementation of the JBI interfaces, supports facilities for different kinds of listeners. The interfaces listed below represent an extract of the listeners available in ServiceMix which is useful for interception.

ExchangeListener This interface has two methods: The 'exchangeAccepted' method is called when a message exchange is accepted by a delivery channel (see section 4.1.1), the 'exchangeSent' method when an exchange is sent by a delivery channel. Thus, for e.g. one execution of the In-Only-MEP, exchangeSent is called three times, as well as exchangeAccepted.

EndpointListener This interface has methods which are called when endpoints are registered or deregistered. They are also called once for each endpoint on startup.

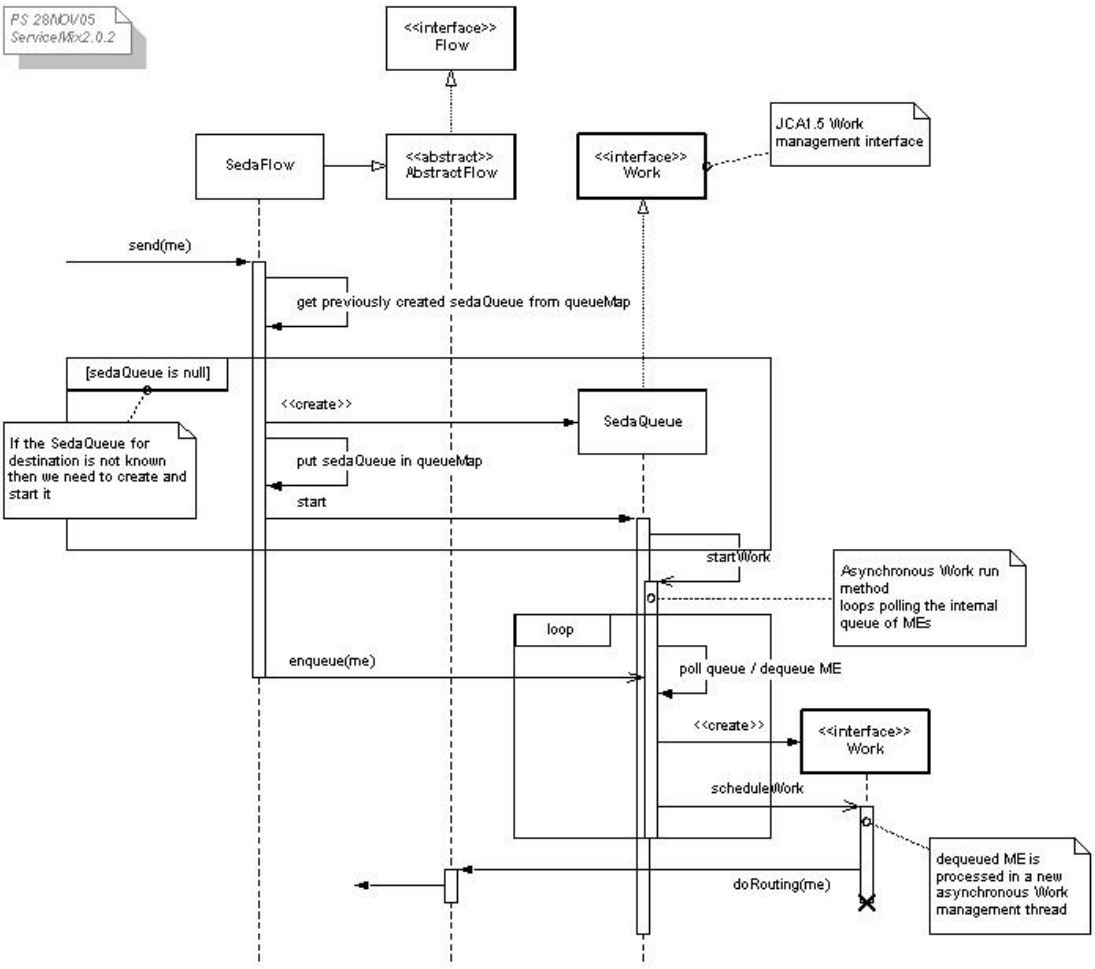

Figure 4.4: Procedure for SEDA flow in ServiceMix [4]

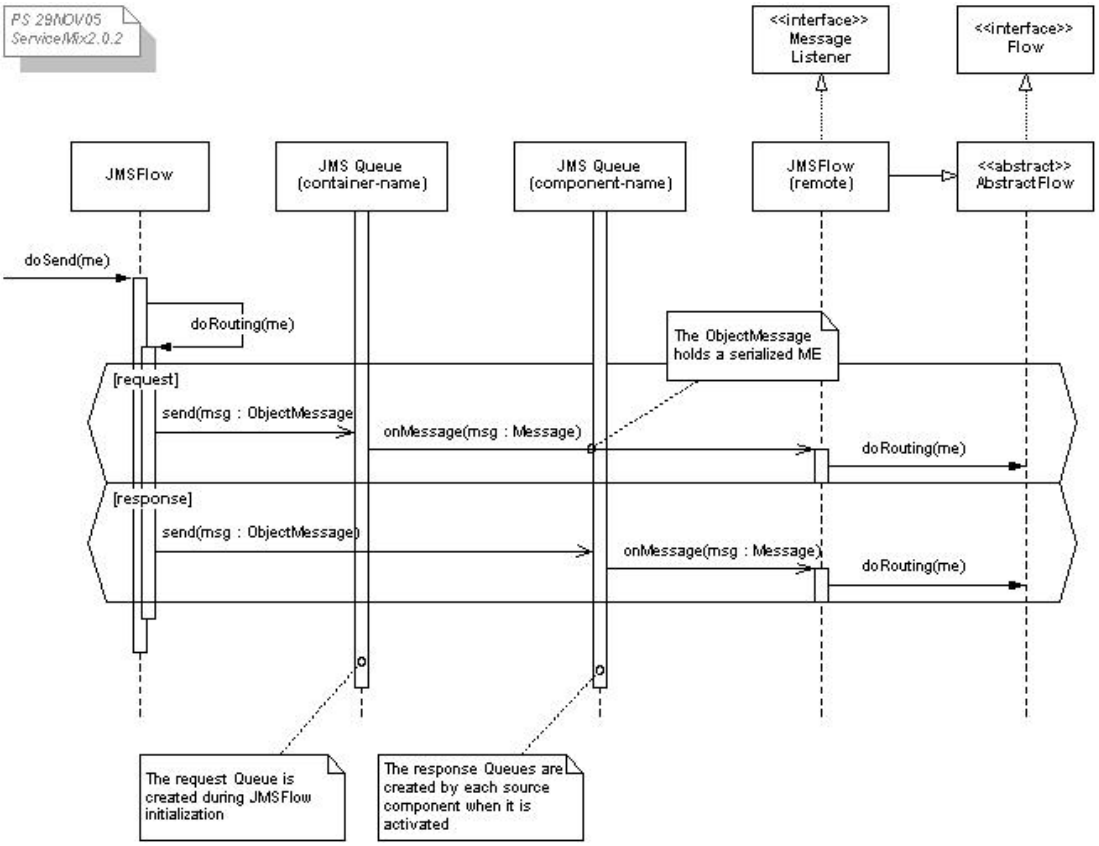

Figure 4.5: Procedure for JMS flow in ServiceMix [4]

ServiceUnitListener As the EndpointListener interface, this interface has methods called when service units are started, stopped, deployed or undeployed. There are similar listener interfaces for service assemblies and components.

These listeners can be attached to the JBI container implementation of ServiceMix. The respective methods of the listeners are then automatically called when appropriate.

The JBI container implementation of ServiceMix holds lists of all relevant entities of the runtime environment, such as registered endpoints or deployed service units. This is useful to get an overall view of the current state, for example to generate the instances for the ontology explained in section 4.2.1, API package.

4.2 Interception in ServiceMix

The implementation of the signaling component prototype written as part of this thesis is presented in this section. Starting from the ServiceMix specific How-Model from chapter 3, it should be capable of covering the interception points marked in the model, or be able to get the same information from another part of the ESB.
The prototype aims at not changing any existing code, but only adding additional services by using the interfaces provided. As it uses the JBI interface, it will later be possible to reuse parts of the code for other JBI based ESBs. However, the listener functionality provided by ServiceMix, which is heavily used in this prototype, will have to be implemented when writing a signaling component for other JBI based ESBs.

4.2.1 Architecture

Figure 4.6 depicts the basic units of the signaling prototype. On the top of the figure, example binding components (BC) and internal services (IS) are shown which communicate through the normalized message router. The MasterAuditor catches the message exchanges on the bus and filters them based on the policies handled by the policy handler. It creates an event per exchange that passed the filter and sends it to a (list of) endpoint(s), depending on the policies that matched the message exchange. In the current version, there are two events sinks available as examples. One writes the events to the file system, the other sends them to a web service that saves them in a database.
The policies can be updated using a SOAP interface (PI). The SOAP requests are handled by an internal service (PolicyBean), which updates the policy handler accordingly.
The second part of the signaling prototype (OntologyAuditor) is designed in order to provide an overview of the current state of the ESB. This includes the (static) structure of the ESB formalized as an ontology. It corresponds to the ServiceMix specific How-Model presented in chapter 3. Further, it provides the actual state of the ESB by

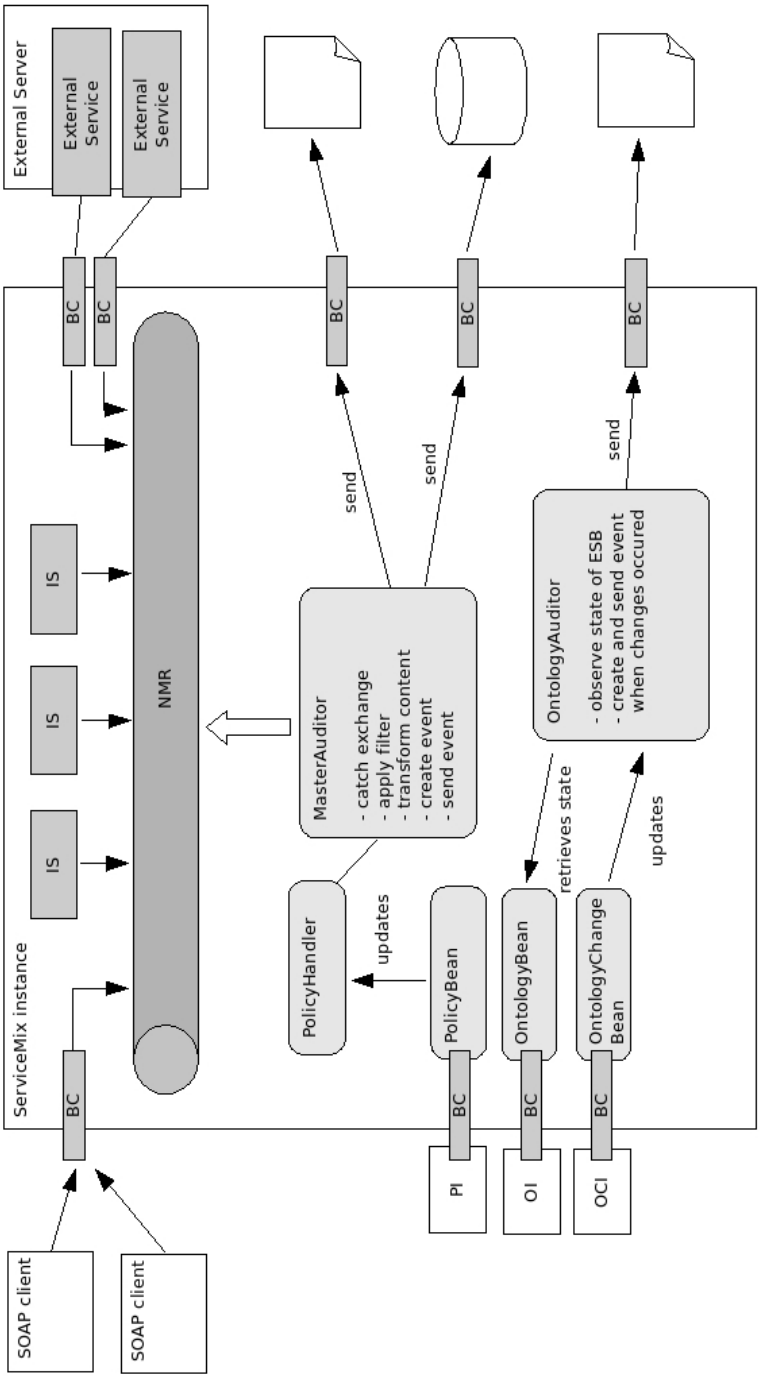

Figure 4.6: Basic architecture of the MasterAuditor

listing all endpoints and services currently present as instances of the ontology.
The OntologyAuditor can be controlled, similar to the MasterAuditor, using a SOAP interface (OCI). These requests are internally handled by the OntologyChangeBean service. Another SOAP interface (OI) is used to get the current state of the ESB formalized as an ontology. Requests sent to this interface are handled by the OntologyBean.
The signaling implementation for ServiceMix is grouped into two packages, which are both subpackages of the already existing package org.apache.servicemix.jbi.audit.

org.apache.servicemix.jbi.audit.master.interception The interception package contains the part of the signaling prototype responsible for catching the messages sent by services and generating events out of them. This part also handles the policies describing which messages have to be observed.

org.apache.servicemix.jbi.audit.master.api The API package contains all classes needed for the ontology interface.

Interception package

Figure 4.7 illustrates the basic classes of the interception package. The class MasterAuditor is the central class. It implements (transitively) the ExchangeListener interface presented in section 4.1.2. As it also implements the InitializingBean interface, the afterPropertiesSet method is called on initialization. This method registers the MasterAuditor instance as a listener on the container. The MasterAuditor passes message exchanges it receives to a dispatcher. The dispatcher keeps an ExchangeHandler instance, which is the starting point for a chain of responsibility. The first instance in this chain is of type PolicyExchangeHandler. It could be used, if necessary, to filter message exchanges that are sent to set e.g. policies and should thus not be taken into account for generating an event. With the current implementation of the filter, this is not needed. The second element of the chain is a MessageExchangeHandler. It parses the message and generates and sends an event to a list of event sinks. It uses a PolicyHandler to determine which message exchanges to observe (filtering), and where to send the events to. The currently available PolicyHandler implementation, FilePolicyHandler, uses the file system for policy persistence. In listing 4.1, an example policy is shown. This policy determines that all messages sent by the endpoint denoted in the EndpointToObserve-Tag have to be observed, and the generated events by sent to the EndpointToSendTo. The policy id is unique. The fields 'direction' and 'blocking' are reserved for enforcement and not used in the prototype.
Another class in the interception package is PolicyBean. It extends ComponentSupport and can thus be started by specifying it in the central configuration file of ServiceMix (servicemix.xml). This is explained in section 4.2.3. The PolicyBean represents the SOAP interface (ESBSignallingLifecycle.wsdl) for setting and deleting policies. It parses the SOAP messages and uses a reference to the MasterAuditor instance to set and delete the policies managed by the PolicyHandler.

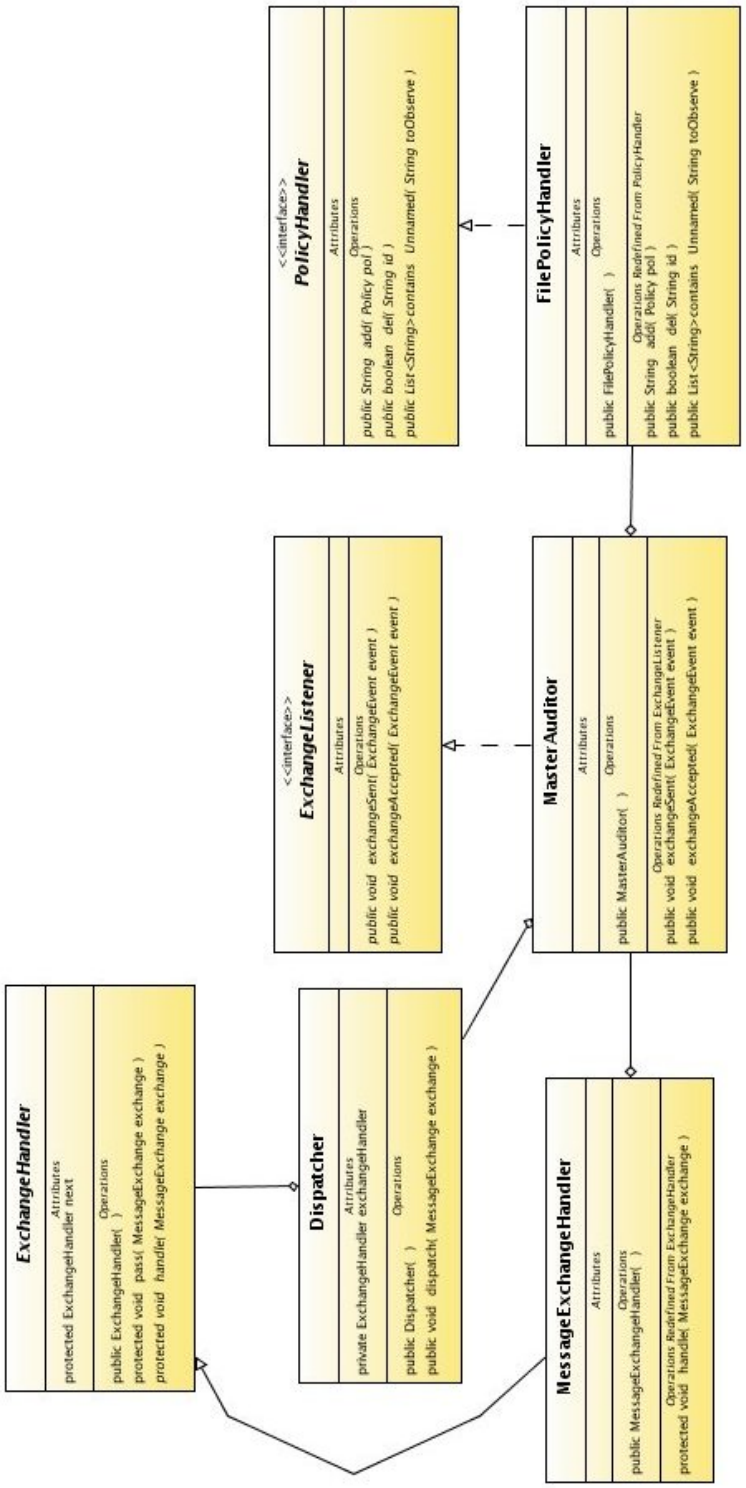

Figure 4.7: Basic classes of package org.apache.servicemix.jbi.audit.master.interception

Listing 4.1: Example policy

```
<esb:addPolicy>
    <Id>policyID</Id>
    <EndpointToObserve>servicename_namespace;
            servicename_localpart;endpointname</EndpointToObserve>
    <EndpointToSendTo>servicename_namespace;
            servicename_localpart;endpointname</EndpointToSendTo>
    <Direction>request</Direction>
    <blocking>false</blocking>
</esb:addPolicy>
```

The class Policy represents a signaling policy, the class MEF an event in the event format specified by MASTER.
Figure 4.8 illustrates the workflow of processing a message exchange. The delivery channel of the service sending a message calls the exchangeSent() method of MasterAuditor[1]. This passes the message exchange to the MessageExchangeHandler. The MessageExchangeHandler first checks if there are policies that match the source endpoint of the exchange. The policies are available in a HashMap, so this is checked in constant time. If there are such policies, an event (appendix A) has to be created. This must be done immediately, as some values of the event may change later (especially the time the event is created). Creating the event consists of setting fields of a MEF object, including the payload (content) of the message. This content is of type javax.xml.transform.Source. So, it is either a DOMSource, a StreamSource or a SAXSource. When the message's source endpoint is the CXF-component, then the content is of type DOMSource. To embed the content in the event, it must be transformed to a String, which is expensive. After serializing the event, it is sent to an event sink. This involves creating a normalized message and setting the event as its content. For this, the event (as String) has to be transformed back to an object of type javax.xml.transform.Source.
Several improvements could make the described process faster:

- The content of the message can be attached to the MEF object as an object of type javax.xml.transform.Source, and the message put in a queue. Several other threads can then transform the content to a String. This implies new problems according to message ordering, especially important for enforcement.

- Sending the events is, depending on which component is used, a bottleneck. If the events are e.g. sent to the filesystem, then one approach to make it faster is sending bunches of events, as this reduces file accesses.

[1]The exchangeSent() method is called four instead of three times as expected for the in-out MEP. This is a known bug, workarounds and patches are available on the ServiceMix website [5]

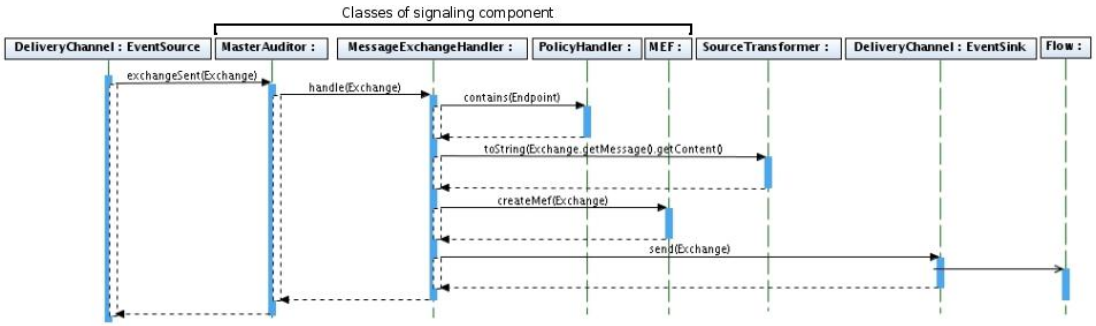

Figure 4.8: Message processing workflow

API package

The basic classes of the API package are depicted in figure 4.9. The class MasterOntology implements (transitively) the interface Ontology, which represents the main interface used by clients that want to obtain information about the current state of the ESB, or the instances of the ontology, respectively (ServiceMix specific How-Model, chapter 3). The instance of MasterOntology also holds a reference to an OntologyObservable. This class implements the java.util.Observable interface. Instances of classes that extend the abstract class OntologyObserver can be registered on the OntologyObservable as listeners, and will be informed in case of a change of the ESB status. The class OntologyEventsSender is a listener on this observable. It sends a MetaDataChangeNotification instance to a list of endpoints.
The class OntologyChangeBean is, similar to PolicyBean of the interception package, used to update the list of endpoints metadata change notifications are sent to via a SOAP interface (sigmetadatachange.wsdl). Another SOAP interface serves the purpose to get the current state of the ESB (Signaling-Metadata.wsdl). An instance of OntologyBean parses these requests and uses the Ontology interface (the instance of MasterOntology implementing this interface respectively) to answer them.

4.2.2 Further implementation steps

The next implementation step for the signaling component is to integrate MXQuery [6] to use its stream processing mechanisms. This was initially planned to be part of this thesis.
For two reasons, I could not implement this step anymore:

- It was planned for the thesis to make a classification of ESBs to see commonalities among them and determine how to use them. It was then demanded by MASTER to extend this specification and create ontologies describing the signaling process, which led to the models presented in chapter 3. When later coming to the integration of ServiceMix into the testbed, the testbed was not yet

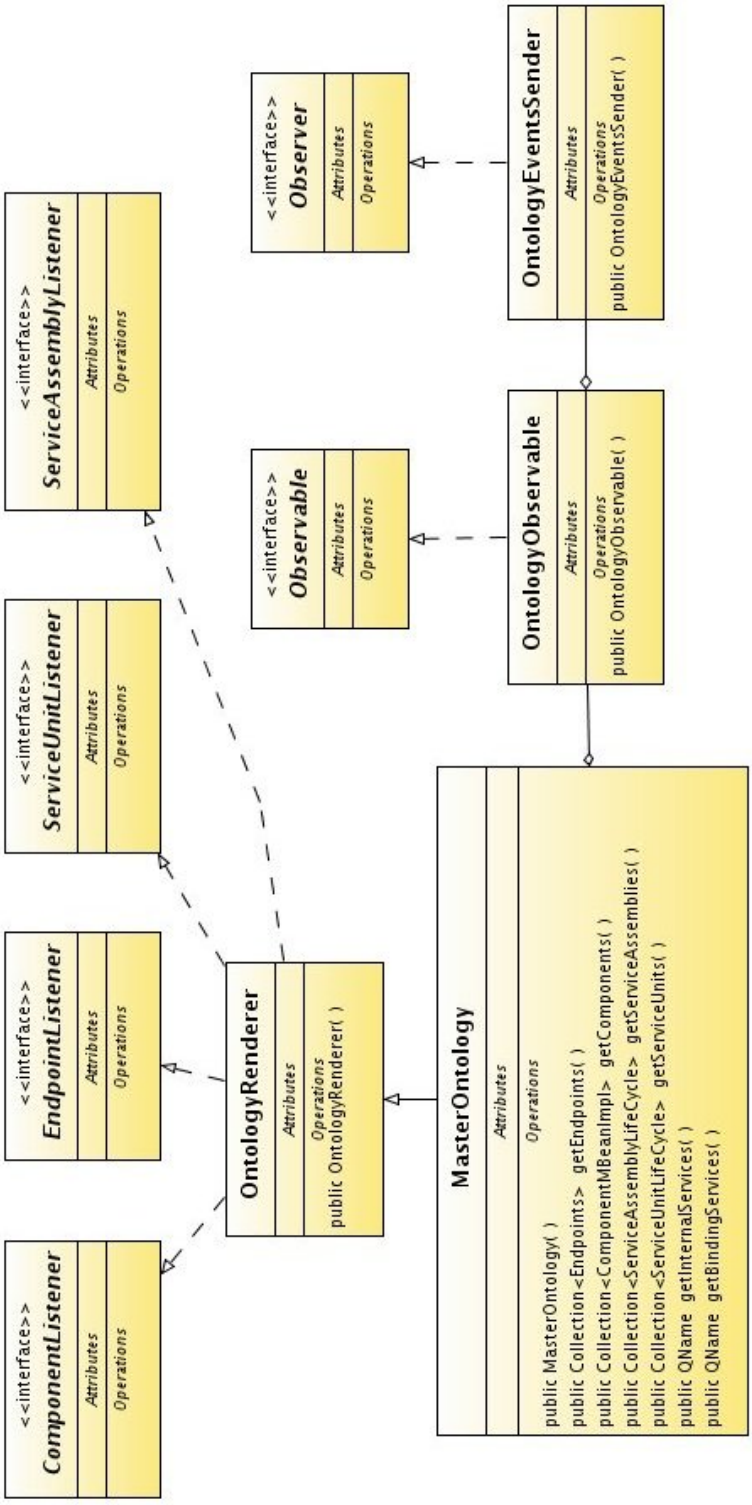

Figure 4.9: Basic classes of package org.apache.servicemix.jbi.audit.master.api

set up by the responsable participant of MASTER.

- Messages created by components are always subtypes of type javax.xml. transform.Source. The CXF component creates messages of type DOMSource. MXQuery currently does not accept this type of Source. This needs to be implemented first.

4.2.3 How to install and use the signaling prototype

For simplicity, the two subpackages of the prototype can be packaged in a jar-file containing the whole org.apache.servicemix.jbi.audit package. Thus, the existing audit-package (apache-servicemix-xxx.jar) can be exchanged with a new jar-file also containing the interception and API subpackages. If the new package has the same name as the old one, no more changes need to be made in the configuration of ServiceMix to integrate the new classes.
However, to start interception, the appropriate entries have to be set in the file servicemix.xml. There are separate entries for the interception part and for the API (ontology part). If required, the beans implementing the SOAP interfaces (PolicyBean, OntologyChangeBean, OntologyBean) can be set there as well.
A complete checklist on how to compile and install a pure ServiceMix instance as well as how to use the prototype and modify the ServiceMix configuration can be found in appendix B.

Chapter 5

Performance

5.1 Goal of performance experiments

The performance experiments' goal is to check how the signaling prototype performs, and which steps of interception are bottlenecks for throughput. This information is then used to determine what needs to be changed, and how a next version of the signaling component needs to be designed. The interception steps, as explained in section 4.2.1, are illustrated in figure 5.1. This diagram is referenced by the experiment descriptions in this chapter.

The most important question concerns message throughput peaks, and how they can be handled. Latency is important for enforcement, where messages are blocked while deciding what should happen.

The test infrastructure is built on the testbed proposed by MASTER. It uses web services installed for this testbed, which are accessed by a soap client via the ESB.

The baseline for the tests is a bare ServiceMix instance with components that implement a SOAP bridge for the web services. These results are compared to performance test results from another test [17]. Then, the impact of the steps performed in order to generate events is evaluated.

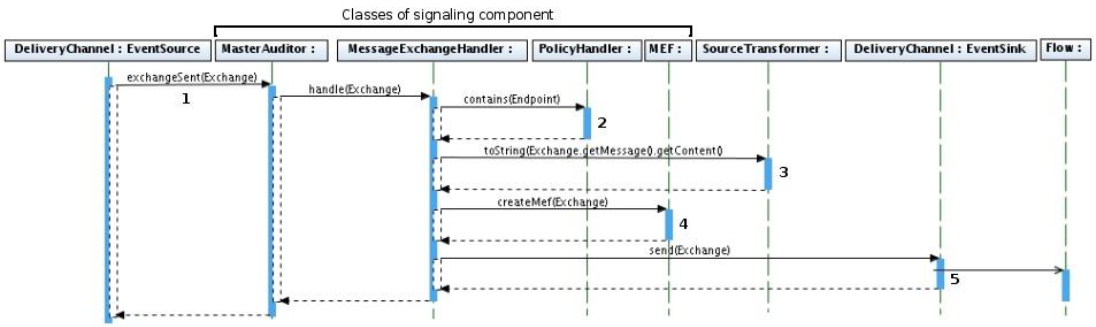

Figure 5.1: Message processing workflow with interception steps

5.2 Test infrastructure

5.2.1 Testbed

To test the signaling prototype, the testbed as proposed by MASTER is used. It consists of three web services that can be accessed via the ESB. For this purpose, six services are deployed on ServiceMix. They use the CXF-binding-component to convert the SOAP requests and responses to the Normalized Message format and back. The configuration of the CXF-components is given in appendix C.1. Two services are needed per web service. One, the consumer service, is on the client side. It converts SOAP requests to Normalized Messages and sends them to the internal endpoint of the second service, the provider. This service converts the message back to a SOAP request and sends it to the external web service. The response is transmitted vice versa. The testbed is illustrated in figure 5.2. It also depicts the basic units of the signaling prototype used for message interception. The six binding services (BC) used for proxying the web services are shown at the top of the figure. The MasterAuditor instance catches the message exchanges and filters them based on the policies managed by the PolicyHandler. The payload of the message is, in the case of a web service brigde, the body of the SOAP request. It has to be transformed to a String to embed it in the event created for the exchange. After the event is converted to a String, it is attached to a newly created message exchange and sent to a (list of) endpoint(s), depending on the policies available for the corresponding source endpoint.

5.2.2 Setting

For testing the performance of the signaling prototype embedded in the testbed as explained in section 5.2.1, Apache Bench [7] is used. Apache Bench is an HTTP server benchmarking tool. It can be used to send bunches of SOAP requests and check if the response is valid. The following options of Apache Bench are used in the tests:

-n The number of requests being sent

-c The number of concurrent connections used to send the requests

-k Perform multiple requests within one HTTP session (keep alive)

-p The XML file containing the SOAP message

The command

```
ab -n 1000 -c 100 -k -p hostrequest.xml http://host:port/HostService
```

thus starts Apache Bench, which sends 1000 requests containing the content of the file hostrequest.xml to the web service on http://host:port/HostService, using 100 concurrent connections, and not closing the HTTP connection.
The tests are performed by sending 5000 requests for warm up, and then sending 5

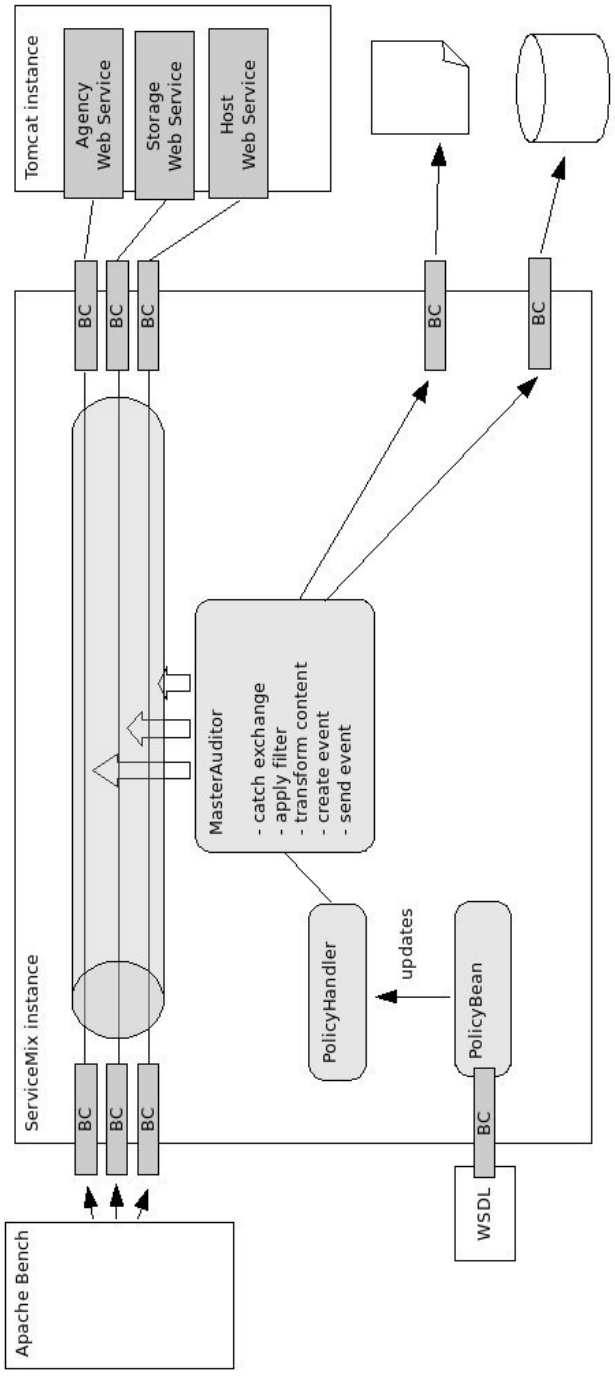

Figure 5.2: Testbed for performance evaluation

times 1000 requests to measure throughput and latency. For throughput measurements, 100 concurrent connections are used, as this leads to the best tradeoff between concurrency benefit and overhead. The requests to measure the latency are sent in sequence (with 1 connection). The final values are then determined by calculating the mean of the five values measured (for the latency test, this correnspponds to the mean of 5000 values).

The following settings have been made on the services, on ServiceMix and on the machine that hosts the ServiceMix instance:

JBI Wrapper The useJBIWrapper attribute of the CXF binding services is set to false. This makes message exchange faster.

Listener All listeners not belonging to signaling are commented in the file servicemix.xml, and thus not started.

Log level The log level is set to INFO. This is done by setting the value for org.apache.servicemix in the file log4j.xml.

Local disc If a persistent flow is used (e.g. JMS flow), ServiceMix must have the possibility to write on a fast, local hard drive.

OS settings On linux, values are set concerning file system limitations. See appendix C.2 for the correct values.

The ESB is running on a linux machine with an Intel Xeon CPU, 2.33GHz, 8 cores and 4096 KB cache. The Java VM is started with 2048 MB initial and maximum heap size.

5.3 Experiments

The first experiment tests the testbed without signaling turned on, and compares throughput and latency of the requests to results of a test performed by WSO2 [17] in june 2008. The second experiment gives an overview of the costs of the whole signaling process. In the following experiments, the signaling steps performed to generate an event are applied one by one and the impact measured. This identifies which steps of signaling have the highest impact on latency and throughput. The last experiments determine the impact of using a persistent flow, and point out heap size and CPU usage for the experiment runs.
All tests are run with two different web services (Host, Agency), and two different SOAP requests per web service. One request is the original one, the second request is extended with comments to simulate a ten times bigger request. The original request for the Host service has a size of 450 bytes, the one for the Agency service 1400 bytes.

Table 5.1: Results for experiment 1: Bare web services

	Host service		Agency service	
	Throughput [req/sec]	**Latency [msec]**	**Throughput [req/sec]**	**Latency [msec]**
bare service, small request message	2844	1.11	3145	1.04
bare service, big request message	1655	1.77	3082	1.44

5.3.1 Experiment: Baseline

The goal of this experiment is to check the testbed. For this purpose, throughput and latency for a bare ServiceMix installation (without any signaling) are compared to throughput and latency results of a test made by WSO2 [17].
To estimate the influence of the processing of the web service request, throughput and latency for the web services are also measured.

Results

In table 5.1, results for throughput and latency for the two web services used without ServiceMix are shown.
Table 5.2 lists throughput and latency results for the baseline experiment (bare ServiceMix). For ServiceMix, only suitable values for throughput are available from the WSO2 tests. The size of the message used for this experiment is 570 bytes. The message size for the 'small message' variant for the Host service is encreased by adding comments to it. As no significant difference was measured, only the results for the tests using the original message (450 bytes) are given.

Interpretation

The latency of the requests sent through the ESB is much higher than the latency for the bare web services. However, it changes depending on the request size. This has to be taken into account when interpreting the results of the further tests. As throughput is anyway higher than with ServiceMix, the web services are not a bottleneck.
The performance of the testbed is worse both for latency and throughput then the performance of the WSO2 testbed. As they used a service with a simulated latency of 10 milliseconds, and the services used in this test have lower latency values, this does not explain the degradation. The WSO2 test uses, however, the ServiceMix' HTTP component to transmit SOAP requests, which is faster. Although, it is proposed to use the CXF component instead, which supports more SOAP features.

Table 5.2: Results for experiment 1: Baseline

	Host service		**Agency service**	
	Throughput [req/sec]	**Latency [msec]**	**Throughput [req/sec]**	**Latency [msec]**
bare ServiceMix, small request message	569	7.07	574	6.81
bare ServiceMix, big request message	567	7.14	547	7.44
	Echo service			
	Throughput [req/sec]	**Latency [msec]**		
ServiceMix in WSO2 test	1380	?		

To prove that the CXF-component is the bottleneck leading to the throughput measured above, another test was performed which used two internal services that directly sent messages to each other. Using the SEDA flow and a payload corresponding to the small request messages, about 155 000 messages are transmitted per second. This corresponds to a throughput of about 52 000 requests per second, as each web service request leads to three Normalized Messages on the bus. With the higher payload (big request message), about 148 000 messages are transmitted. Using the JMS flow decreases these numbers significantly to 2300 and 1800, respectively.

5.3.2 Experiment: Signaling

This experiment tests the impact of the whole signaling process (steps 1 to 5 in figure 5.1) on throughput and latency as an overview.

Results

Table 5.3 and table 5.4 list throughput and latency values for the whole signaling process depending on the web service used and the message size, and compares them to the numbers for a bare ServiceMix installation (from baseline experiment).
Figure 5.3 shows a chart displaying the throughput numbers measured in the experiments for the Agency service. For this service, the difference in size between the big and the small request message is more significant. The x-axis displays the signaling steps as in the experiments, the y-axis the throughput in requests per second. In figure

Table 5.3: Results for experiment 2: Whole signaling process, throughput [req/sec]

	Host service		**Agency service**	
	bare SM	signaling	bare SM	signaling
small request message	569	196	574	188
big request message	567	174	547	106

Table 5.4: Results for experiment 2: Whole signaling process, latency [msec]

	Host service		**Agency service**	
	bare SM	signaling	bare SM	signaling
small request message	7.07	14.37	6.81	13.92
big request message	7.14	18.62	7.44	19.58

5.4, the y-axis displays the latency in milliseconds. The intermediate steps measured in the following experiments are already sketched.

Interpretation

The results of the above experiment show that the signaling process has a big impact on throughput and latency. The values depend on the messages size, which is more significant for the Agency service, where the messages are bigger. The following experiments will identify the most expensive steps.

5.3.3 Experiment: Intermediate steps

This experiments goal is to understand the impact on throughput and latency of the individual steps taken to catch and transform a message (steps 1 to 3 in figure 5.1) and generate and send the event to a file endpoint (steps 4 and 5 in figure 5.1). The services and request messages used are the same as in the last experiment.

Results

Table 5.5 lists the results for throughput of the requests sent. Column 'listener' contains the values measured when MasterAuditor is started and registered as listener (step 1 in figure 5.1), and thus it's exchangeAccepted() method called on a message exchange. Column 'filter' depicts the values when the filter is turned on, and column

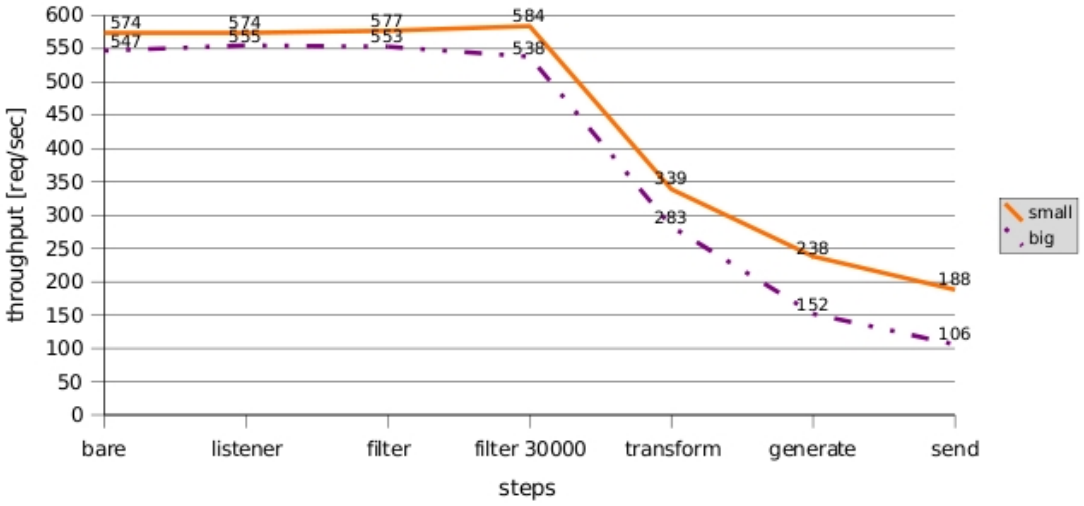

Figure 5.3: Results summary: Throughput depending on message size and signaling step

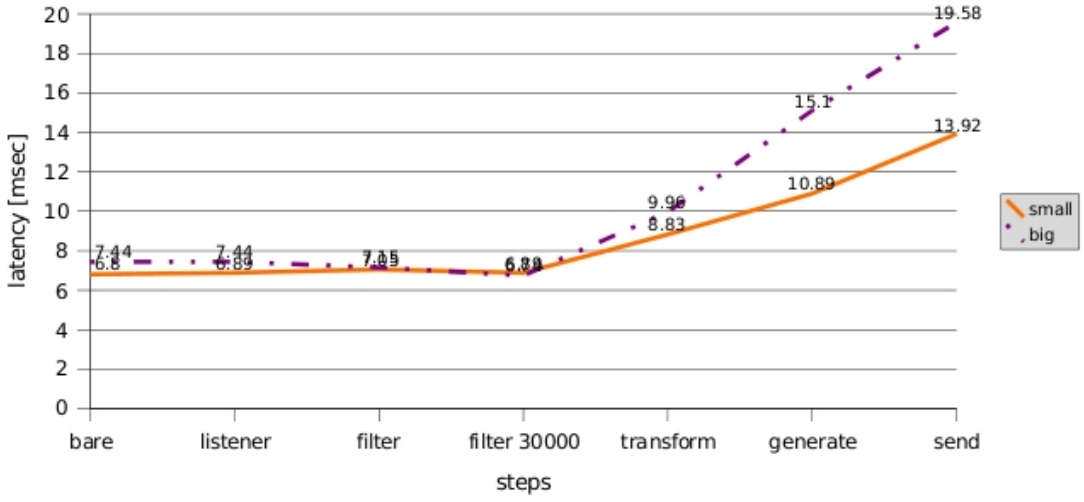

Figure 5.4: Results summary: Latency depending on message size and signaling step

Table 5.5: Results for experiment 3: Content transformation, throughput [req/sec]

	listener	filter	filter 30000	transform	generate	send
Host service						
small request message	588	591	586	342	239	196
big request message	568	579	568	321	215	174
Agency service						
small request message	574	577	584	339	238	188
big request message	555	553	538	283	152	106

'filter 30000' when the filter contains a huge amount of policies (30 000 policies, step 2 in figure 5.1). The next column, 'transform', shows the impact of transforming the message content from a DOMSource to a String (step 3 in figure 5.1). The values in column 'generate' are measured when the event object is serialized to a String, and the last column ('send') when the event is sent to a file endpoint.
Table 5.6 is organized alike, but shows the results for latency.

Interpretation

For the first three columns, no significant difference compared to the bare ServiceMix installation is found. This implies that attaching the listener and filtering (on one property) does not affect performance significantly. Even when a lot of policies are registered, filtering does not consume more ressources, as the policies are accessible in constant time. Transformation of the messages, though, is much more expensive, and is dependent on the message size. Generating the event again needs a message transformation (from String to StreamSource), and saving the event has filesystem limitations. For the design of the signaling component, this means, that it should be checked if it is possible to delay transformation of the content. As the message is never changed after creation, from the view of monitoring, this would be a valid solution. For enforcement, however, the message content must be available instantly, as other messages might be blocked. Sending to the file component could be made much faster be allowing to store bunches of events. For enforcement, this solution would introduce new challenges.

Table 5.6: Results for experiment 3: Content transformation, latency [msec]

	listener	filter	filter 30000	transform	generate	send
Host service						
small request message	7.02	6.81	6.32	8.39	11.88	14.37
big request message	7.33	7.12	6.35	9.44	12.85	18.62
Agency service						
small request message	6.89	7.05	6.89	8.83	10.89	13.92
big request message	7.44	7.15	6.74	9.96	15.10	19.58

5.3.4 Experiment: Message persistence

The persistent JMS queues ServiceMix uses for the JMS flow could be used to achieve reliable message transport. This experiment points out the effects of using JMS instead of the in-memory flow.
For these tests, the bare ServiceMix installation without MasterAuditor activated is tested as well as the full signaling process.

Results

Table 5.7 shows the throughput results for both web services using the persistent JMS queues and compares it to the values determined when using the in-memory queues (SEDA-flow).
Table 5.8 lists the same results for latency.

Interpretation

The results show that if persistency is required, it will have a major impact on the performance. Especially the message size degrades the results. This is best shown with the results for the Agency service, where the request message is about three times bigger than for the Host service.

Table 5.7: Results for experiment 4: in-memory and persistent queueing, throughput [req/sec]

	Host service		**Agency service**	
	in-memory	persistent	in-memory	persistent
bare SM, small request message	569	313	574	270
bare SM, big request message	567	290	547	176
signaling, small request message	196	96	188	74
signaling, big request message	174	68	106	45

Table 5.8: Results for experiment 4: in-memory and persistent queueing, latency [msec]

	Host service		**Agency service**	
	in-memory	persistent	in-memory	persistent
bare SM, small request message	7.07	10.82	6.81	12.49
bare SM, big request message	7.14	12.30	7.44	14.83
signaling, small request message	14.37	30.77	13.92	33.84
signaling, big request message	18.62	39.39	19.58	48.61

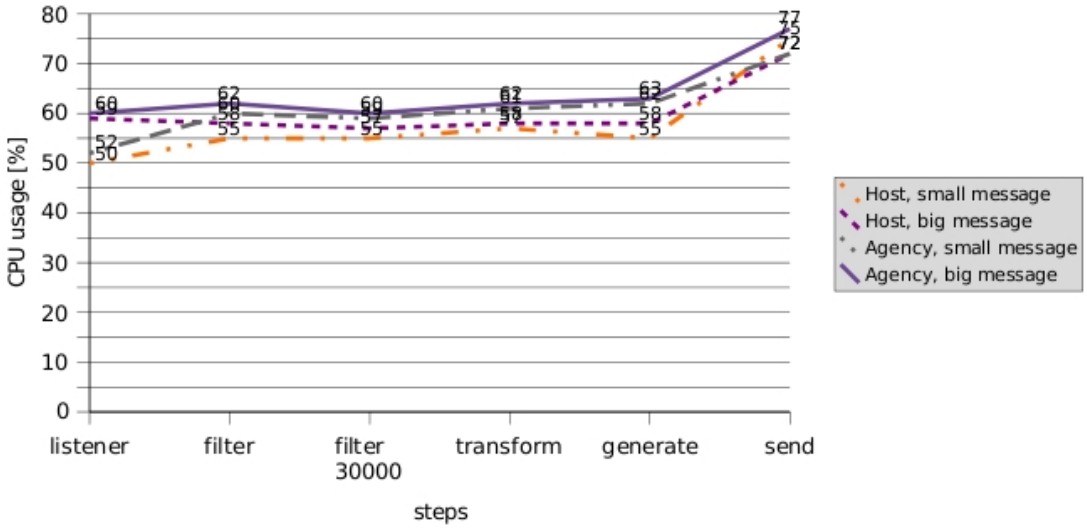

Figure 5.5: Results for experiment 5: CPU usage during experiments [%]

5.3.5 Experiment: Heap size and CPU usage

In this section, Java VM heap size and CPU usage values measured during the experiments are presented.

Results

Figure 5.5 depicts CPU usage depending on the message size and the signaling steps performed. The x-axis shows the steps, the y-axis the CPU usage in percent.
Figure 5.6 shows the same for the Java VM heap size. The y-axis displays the heap size in MegaBytes.

Interpretation

Figure 5.5 shows that the CPU usage only increases significantly when the file component sends the events to the file system. There is no significant dependency on the message size.
For the heap size, the dependency on the message size is very significant. Main memory is consumed when sending the events, as they are transmitted to the file component using a flow with an in-memory queue.

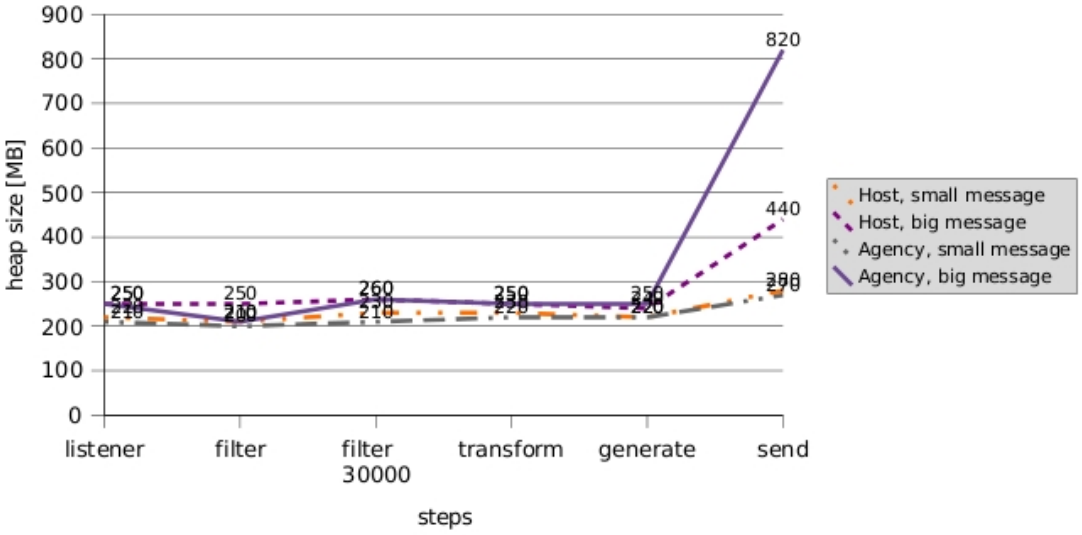

Figure 5.6: Results for experiment 5: Heap size during experiments [MB]

Chapter 6

Conclusion

This thesis presents the design, implementation and evaluation of a protoype for the signaling process required by the MASTER project.
For this purpose, an overview of the current ESB landscape is given. As this landscape turned out to be very heterogeneous, a more concrete classification was required. The MASTER project demanded the classification to be in the form of ontologies. The models thus describe the ESB landscape, the signaling process and the mapping from concrete ESB interfaces to the signaling interface as ontologies.
The prototype is built on the basis of the ESB implementation Apache ServiceMix. It is an ESB based on the JBI specification, which represents a large group of ESBs. The prototype performs the individual signaling steps to show their practibability.
The performance evaluation revealed the weaknesses of the current design. Messages have to be transformed from different XML-representations to Strings and back, which is very expensive. Also, sending the events to an external service (as well as to the file system) is expensive, as it includes message transformation again and has limitations from the external system. In the following versions of the protoype, some of these steps can be parallelized, by still complying with constraints of monitoring and enforcement.
In a future step, MXQuery is integrated in the prototype to test its applicability for processing the events. More complex filtering and processing is then possible, by using XQuery's windowing features. The current MXQuery version does not accept the XML representation used by the ESB in the testbed. This feature has to be implemented first.

Bibliography

[1] http://www.ibm.com/developerworks/web/library/wa-soaesb/.

[2] http://sunset.usc.edu/GSAW/gsaw2007/s7/hohwald.pdf.

[3] http://servicemix.apache.org/servicemix-cxf-bc.html.

[4] http://servicemix.apache.org/how-stuff-works.html.

[5] https://issues.apache.org/activemq/browse/SM-1712.

[6] http://www.dbis.ethz.ch/research/current_projects/MXQuery.

[7] http://httpd.apache.org/docs/2.0/programs/ab.html.

[8] Apache ServiceMix. http://servicemix.apache.org.

[9] Apache Synapse. http://synapse.apache.org/.

[10] BEA Aqualogic. http://www.oracle.com/bea/index.html?CNT=index.htm&FP=/content/products/aqualogic/.

[11] Java Business Integration specification. http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html and http://jcp.org/en/jsr/detail?id=312.

[12] MASTER: Managing Assurance, Security and Trust for sERvices. http://www.master-fp7.eu/.

[13] MuleESB. http://www.mulesource.org/display/MULE/Home.

[14] Petals ESB. http://petals.ow2.org/.

[15] Progress Sonic ESB. http://www.sonicsoftware.com/products/sonic_esb/index.ssp.

[16] Sun OpenESB. https://open-esb.dev.java.net/.

[17] WSO2 ESB Performance testing, round 3. http://wso2.org/library/3740.

[18] Ron Ten-Hove. JBI Components: Part 1 (Theory). Technical report, Sun Microsystems, 2006.

[19] Volkmar Lotz, Emmanuel Pigout, Dr. Peter M. Fischer, Prof. Donald Kossmann, Prof. Dr. Fabio Massacci, Dr. Alexander Pretschner. Towards Systematic Achievement of Compliance in Service-oriented architectures: The MASTER approach, 2008. Available at: http://www.wirtschaftsinformatik.de/index.php;do=show/site=wi/sid=88285986849bd5ffd02df4373318728/alloc=12/id=2285.

Appendix A

Example MEF event

Example event in Master Event Format: the payload of the MasterEvent element contains the request message (<in>) and the response message (<out>).

```
<cbe:CommonBaseEvent xmlns:cbe="http://www.ibm.com/AC/commonbaseevent1_1"
 xmlns:mef="http://www.master-fp7.eu/mastercommonbaseevent1_0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="cbe.xsd ">
<sourceComponentId application="ServiceMix" location="10.110.64.5"
 subComponent="subComp" component="ServiceMix#ServiceEndpoint[service=
 {http://www.iaas.uni-stuttgart.de/master/cesce/agency}Agency,endpoint=
 AgencyHttpSoap11Endpoint]#AgencyHttpSoap11Endpoint" locationType="IPV4"
 componentIdType="idType"/><reporterComponentId application="ServiceMix"
 location="10.110.64.5" subComponent="subComp" component=
 "ServiceMix ESB Signaling service for MASTER" locationType="IPV4"
 componentIdType="idType"/><situationInformation creationTime=
 "2009-03-13T17:21:21"/>
<mef:MasterEvent xsi:schemaLocation="mef.xsd">
<payload><metadata/><message>
<in><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body><axis2ns2:informationRequest xmlns:axis2ns2=
 "http://www.iaas.uni-stuttgart.de/master/cesce/agency" xmlns=
 "http://www.iaas.uni-stuttgart.de/master/cesce/agency" xmlns:agy=
 "http://www.iaas.uni-stuttgart.de/master/cesce/agency">
    <companyName>ACME</companyName>
    <information>Elvira</information>
 </axis2ns2:informationRequest></soap:Body></soap:Envelope></in>
 <out><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body><ns:informationRequestResponse xmlns:ns=
 "http://www.iaas.unistuttgart.de/master/cesce/agency">
<ns:return>ok</ns:return></ns:informationRequestResponse></soap:Body>
</soap:Envelope></out>
</message></payload>
<trustInformation> </trustInformation><generalProperties/>
</mef:MasterEvent>
</cbe:CommonBaseEvent>
```

Appendix B

ServiceMix configuration

B.1 Installation checklist

Bare ServiceMix:

- Download the binary distribution of ServiceMix (apache-servicemix-3.3.zip from: http://servicemix.apache.org/download.html)
- Unzip it to a local folder (%SERVICEMIX_ROOT%)
- Run ServiceMix
- Add wsdl files from ServiceMixMaster.zip to %SERVICEMIX_ROOT%
- add cfx-sa.jar to hotdeploy folder

The wsdl files of the services (i.e. http://localhost:8192/AgencyConsumer/?wsdl) should be accessible now.
Making ServiceMix MASTER-aware:

- Shut down ServiceMix
- Add jaxen.jar to %SERVICEMIX_ROOT%/lib
- Replace servicemix-audit-xxx.jar in %SERVICEMIX_ROOT%/lib by the MASTER-aware version of it
- Replace servicemix.xml by the MASTER-aware version
- Run ServiceMix
- Add the corresponding wsdl files for policy manager service (ESBSignallingLifecycle.wsdl) to %SERVICEMIX_ROOT%, adjust soap-address in wsdl if necessary

- Copy the policy manager component to the hotdeploy folder (policy-sa-1.0-SNAPSHOT.jar)
- Repeat the last two steps for:
  - Ontology interface: Signaling-Metadata.wsdl, ontology-sa.jar, http://0.0.0.0:8192/Signaling-MetadataService/?wsdl
  - Ontology change interface: sigmetadatachange.wsdl, ontology-change-sa.jar, http://0.0.0.0:8192/sigmetadatachange/?wsdl
- To set up a collector for generated events there are two options:
  - Writing them to the file system: copy filesaver-sa-xxx.jar to the hotdeploy folder
  - Sending them to an event logger service: copy Event.wsdl to %SERVICEMIX_ROOT%, copy eventLog-sa-XX.jar to hotdeploy
- To set up a collector for generated events from the ontology (e.g. endpoint unregistered): copy ontology-filesaver-sa.jar to hotdeploy (saves events to the filesystem)
- Use retrieveSourceInstances operation in Signaling-MetadataService to get all registered endpoints and services
- Use addPolicy operation in policyManager service to bind listener to some active endpoints
- Use addChangeListener operation in sigmetadatachange to add listener for endpoint or service events

B.2 servicemix.xml configuration file

Specifying active flows:

```
<sm:broker>
  <sm:securedBroker authorizationMap="#authorizationMap">
    <sm:flows>
      <sm:sedaFlow />
      <sm:jmsFlow jmsURL="${activemq.url}" />
        <sm:jcaFlow connectionManager="#connectionManager"
                    jmsURL="${activemq.url}" />
    </sm:flows>
  </sm:securedBroker>
</sm:broker>
```

Adding beans for signaling:

```
<sm:activationSpecs>
    <sm:activationSpec componentName="ESBSignallingLifecycleBean"
            service="policy:ESBSignallingLifecycleBean"
          endpoint="ESBSignallingLifecycleBean">
            <sm:component>
                    <bean class=
       "org.apache.servicemix.jbi.audit.master.interception.PolicyBean">
                    </bean>
            </sm:component>
    </sm:activationSpec>
    <sm:activationSpec componentName="ESBOntologyBean"
            service="ontology:ESBOntologyBean" endpoint="ESBOntologyBean">
            <sm:component>
                    <bean class=
             "org.apache.servicemix.jbi.audit.master.api.OntologyBean">
                    </bean>
            </sm:component>
    </sm:activationSpec>
    <sm:activationSpec componentName="ESBOntologyChangeBean"
            service="ontology:ESBOntologyChangeBean"
                 endpoint="ESBOntologyChangeBean">
            <sm:component>
                    <bean class=
       "org.apache.servicemix.jbi.audit.master.api.OntologyChangeBean">
                    </bean>
            </sm:component>
    </sm:activationSpec>
</sm:activationSpecs>
```

Appendix C

Testbed settings

C.1 CXF configuration

xbean.xml file for cxf-consumer services:

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:agen="http://www.iaas.uni-stuttgart.de/master/cesce/agency"
       xmlns:host="http://www.iaas.uni-stuttgart.de/master/scenarios/
                                                              cesce/host"
       xmlns:stor="http://www.iaas.uni-stuttgart.de/master/cesce/Storage/"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http:// http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/http/1.0
         http://servicemix.apache.org/schema/servicemix-http-3.2.2.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <cxfbc:consumer
                  targetService="agen:Agency"
                  targetEndpoint="AgencyHttpSoap11Endpoint"
                  wsdl="/myAgency.wsdl"
                  useJBIWrapper="false" />

  <cxfbc:consumer
                  targetService="host:Host"
                  targetEndpoint="HostHttpSoap11Endpoint"
                  wsdl="/myHost.wsdl"
                  useJBIWrapper="false" />

  <cxfbc:consumer
                  targetService="stor:Storage"
                  targetEndpoint="StorageHttpSoap11Endpoint"
                  wsdl="/myStorage.wsdl"
                  useJBIWrapper="false" />
</beans>
```

xbean.xml file for cxf-provider services:

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:agen="http://www.iaas.uni-stuttgart.de/master/cesce/agency"
       xmlns:host="http://www.iaas.uni-stuttgart.de/master/scenarios/
                                                              cesce/host"
       xmlns:stor="http://www.iaas.uni-stuttgart.de/master/cesce/Storage/"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/http/1.0
         http://servicemix.apache.org/schema/servicemix-http-3.2.2.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <cxfbc:provider service="agen:Agency"
                  endpoint="AgencyHttpSoap11Endpoint"
                  wsdl="Agency.wsdl"
                  useJBIWrapper="false" />

  <cxfbc:provider service="host:Host"
                  endpoint="HostHttpSoap11Endpoint"
                  wsdl="Host.wsdl"
                  useJBIWrapper="false" />

  <cxfbc:provider service="stor:Storage"
                  endpoint="StorageHttpSoap11Endpoint"
                  wsdl="Storage.wsdl"
                  useJBIWrapper="false" />
</beans>
```

C.2 Operating system settings

Additional settings in file /etc/sysctl.conf on linux:

```
net.ipv4.ip_local_port_range = 1024 65535
net.ipv4.tcp_fin_timeout = 30
fs.file-max = 2097152
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
```

Additional settings in file /etc/sysctl.conf on linux:

```
/etc/security/limits.conf
* soft nofile 4096
* hard nofile 65535
```